

Generating and Searching Families of FFT Algorithms*

Steve Haynal
SofterHardware

steve@softerhardware.com

Heidi Haynal

*Department of Mathematics and Computer Science
Walla Walla University*

heidi.haynal@wallawalla.edu

Abstract

A fundamental question of longstanding theoretical interest is to prove the lowest exact count of real additions and multiplications required to compute a power-of-two discrete Fourier transform (DFT). For 35 years the split-radix algorithm held the record by requiring just $4n \log_2 n - 6n + 8$ arithmetic operations on real numbers for a size- n DFT, and was widely believed to be the best possible. Recent work by Van Buskirk and Lundy demonstrated improvements to the split-radix operation count by using multiplier coefficients or “twiddle factors” that are not n^{th} roots of unity for a size- n DFT.

This paper presents a Boolean Satisfiability-based proof of the lowest operation count for certain classes of DFT algorithms. First, we present a novel way to choose new yet valid twiddle factors for the nodes in flowgraphs generated by common power-of-two fast Fourier transform algorithms, FFTs. With this new technique, we can generate a large family of FFTs realizable by a fixed flowgraph. This solution space of FFTs is cast as a Boolean Satisfiability problem, and a modern Satisfiability Modulo Theory solver is applied to search for FFTs requiring the fewest arithmetic operations. Surprisingly, we find that there are FFTs requiring fewer operations than the split-radix even when all twiddle factors are n^{th} roots of unity.

KEYWORDS: *Fast Fourier Transform, FFT, SMT-Solver, Boolean Modeling*

1. Introduction

In 1965 Cooley and Tukey[14] started a revolution in digital signal processing when they introduced their fast Fourier transform algorithm (FFT). Their FFT required only $O(n \log n)$ addition and multiplication floating-point operations on real numbers, or FLOPs, rather than the $O(n^2)$ FLOPs required to directly compute a discrete Fourier transform. Although discovered previously[25][50], it was Cooley and Tukey’s timing, which coincided with the beginning of widespread use and availability of digital computers, that led to its success. The FFT and related algorithms have now found a wide range of application, including electroacoustic music, audio signal processing, medical imaging, image processing, pattern recognition, computational chemistry, error correcting codes, spectral methods for PDEs and harmonic analysis[5][50].

After the FFT’s introduction, there was considerable work on further lowering the FLOP count. This was of particular interest since addition and especially multiplication were expensive with the computer hardware available at that time. One result that stands out

* August 30, 2011, preprint. To appear in the Journal of Satisfiability, Boolean Modeling and Computation.

is the work done by Yavne[56] in developing an initial split-radix[18][54] algorithm with a $4n \log_2 n - 6n + 8$ FLOP count for a size- n FFT where n is some power of two, $n = 2^m$. Other important work minimized the number of multiplications but not the total arithmetic complexity[28][29][17][55]. In 2004 Lundy and Van Buskirk[39] demonstrated improvements to the split-radix operation count by using constant complex-value multiplier coefficients or **twiddle factors** that are not n^{th} roots of unity for a size- n DFT. Frigo and Johnson[32] generalized Van Buskirk's pioneering work in the context of optimizing the conjugate-pair split-radix algorithm[33]. Bernstein[1] then described Johnson's algorithm, which is distinct from Van Buskirk's, in terms of algebraic twisting and named it the tangent FFT. In this paper, we refer to Johnson's algorithm and Bernstein's variation of it, differing only in decimation in time versus frequency, as the **tangent FFT**. Van Buskirk's algorithm and the tangent FFT exhibits a modest ($\sim 5.6\%$) reduction in FLOP count when compared to the split-radix, requiring roughly $\frac{34}{9}n \log_2 n$ operations rather than the previous $4n \log_2 n - 6n + 8$.

This paper presents a proof of the lowest FLOP count for certain classes of DFTs. It is beyond the scope of this paper to consider all possible DFTs in our proof. Instead, we focus on the common power-of-two complex FFTs and the flowgraphs[10] implied by them. This scope still includes a rich set of FFTs as our experiments confirm what others have seen[18]; common power-of-two complex FFTs (radix-2, radix-4, decimation-in-time or decimation-in-frequency split-radix, conjugate split-radix, classic or any twisted) all exhibit the same flowgraph structure (they are graph isomorphisms) but have different twiddle factors assigned to the flowgraph nodes. Furthermore, we restrict our scope to FFTs where twiddle factors are n^{th} root of unity. This excludes Van Buskirk's algorithm and the tangent FFT, but we still can show that other algorithms with lower FLOP count than the traditional split-radix exist.

In 1962, a few years before Cooley and Tukey introduced their FFT, Davis *et al.* developed a machine program for theorem proving[15], now referred to as the Davis–Putnam–Logemann–Loveland or DPLL algorithm, which is still at the core of modern Boolean Satisfiability or SAT solvers. In the past decade, several advances and refinements to DPLL have made it practical for larger problems[52][44][21]. New conflict-driven clause-learning (CDCL) SAT solvers, which incorporate these recent advances, are now commonly used in industry to verify hardware and software correctness. Current SAT research benefits from industrial sponsorship and an active community, which organizes conferences and competitions, creates challenge problems, and defines problem formats[38][40][49]. Recently, Satisfiability Module Theories (SMT) generalize SAT beyond binary variables to incorporate higher-level theories such as bitvectors, lists and arrays[47][49]. SMT solvers range from those that simply reduce a higher-level theory to Boolean logic for a SAT solver, to those that extend the core decision procedure to accommodate higher-level theories.

In this paper, we apply a modern SMT solver to find a lowest FLOP count algorithm for the class of FFTs considered. First, we present a novel way to choose new yet valid twiddle factors for the nodes in a FFT flowgraph. This technique is more general and leads to a richer solution space than the twisting[1][42], an algebraic way to correctly change the value of twiddle factors. This solution space of FFTs is cast as a SAT problem using quantifier-free formulas over the theory of fixed-size bitvectors, specified in SMT-LIB format[49], and searched with existing SMT solvers[7][20][8][21]. After applying partitioning techniques, we

are able to find 6616 FLOP count algorithms for size-256 FFTs, and 15128 FLOP count algorithms for size-512 FFTs. These numbers are lower than traditional split-radix, 6664 for size-256 and 15368 for size-512, but not as low as achieved by Van Buskirk's algorithm[39] or the tangent FFT[32][1], 6552 for size-256 and 15048 for size-512, due to our constraint that complex twiddle factors must have modulus one, an absolute value of one.

Although we supply code for a witness size-256 FFT requiring fewer operations than a traditional split-radix[27], we are not addressing algorithm design in this paper. An objective to minimize FLOP count is primarily academic given the capabilities of modern computing hardware. We use it only as a well-defined and widely-understood objective to introduce and demonstrate the power of our formulation and search. We believe the ideas presented in this paper can be used to do FFT algorithm design where twiddle factors are not all n^{th} roots of unity, specific hardware is targeted, or other objectives such as overall performance or accuracy are pursued, but these are the topics of a future paper.

This paper continues with an introduction to the DFT, with emphasis on defining concepts central and unique to this paper. In Section 2, we present a FFT flowgraph representation for generating a family of FFTs. This formulation of the solution space is tailored so that it can be easily cast as a SMT problem. Section 3 introduces a first SMT problem formulation and then develops symmetry reduction and partitioning ideas which allows us to solve larger problems. Finally, we conclude with discussion of our experiments and results, application to FFT algorithm design, and future work.

1.1 Definitions

The DFT (discrete Fourier transform) is a specific kind of Fourier transform whose input is a sequence of numbers instead of a function. The sequence of numbers is often obtained by sampling a continuous function. Throughout this paper, let $n = 2^m$, let $i^2 = -1$, and let ω_n represent the complex n^{th} root of unity $e^{-i\frac{2\pi}{n}}$. The n -tuple of complex numbers $(a_0, a_1, a_2, \dots, a_{n-1})$ is transformed by the DFT into another n -tuple of complex numbers $X(k)$ according to the formula

$$X(k) = \sum_{j=0}^{n-1} a_j \omega_n^{jk}.$$

It is well-known that the complex size- n DFT is a linear operator on \mathbb{C}^n and can be represented as multiplication by an $n \times n$ Vandermonde matrix. For our purposes, it is better to identify the entries of the n -tuple with the coefficients of the polynomial

$$f(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} \in \mathbb{C}[x].$$

Then computing the DFT for a given n -tuple is equivalent to evaluating the polynomial f at each of the n^{th} roots of unity ω_n^k , for $k = 0, 1, 2, \dots, n-1$. That is, $X(k) = f(\omega_n^k)$. So each output value of the DFT is a weighted sum of the a_j , where the weight of a_j in $X(k)$ is ω_n^{jk} .

When an FFT is used to compute a size- n DFT, with twiddle factors of modulus one, the product of all the twiddle factors applied to a_j in the computation of $X(k)$ equals the weight ω_n^{jk} . We'd like to keep track of the accumulated weight on any given a_j through all of the intermediate FFT results. To do this, we employ the polynomial view introduced

by Fiduccia in [23] and elaborated by Bernstein in [1] and Burrus in [11]. Associating the input to a polynomial of degree $n - 1$ with coefficients a_j means that an intermediate FFT result is associated to a polynomial of lower degree whose coefficients are weighted sums of the a_j . For example, when $n = 8$ and

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7,$$

two of the intermediate results of the radix-2 FFT are

$$f \bmod(x^2 - \omega_8^0) = (a_0 + a_2 + a_4 + a_6) + (a_1 + a_3 + a_5 + a_7)x$$

and

$$\begin{aligned} f \bmod(x^2 - \omega_8^4) &= (a_0 - a_2 + a_4 - a_6) + (a_1 - a_3 + a_5 - a_7)x \\ &= (a_0 + a_2\omega_8^4 + a_4 + a_6\omega_8^4) + (a_1 + a_3\omega_8^4 + a_5 + a_7\omega_8^4)x. \end{aligned}$$

Each of the coefficients in the two linear polynomials above is represented by a node in a flowgraph, which we describe in the next section. First, we define some characteristics of these coefficients.

Definition 1.1. The *base* of a coefficient is the a_j of lowest index that appears in the weighted sum comprising that coefficient.

Definition 1.2. The *stride* of a coefficient is the integer difference between the indices of any two successive a_j in the weighted sum, when the terms of the sum are written with the indices in strictly increasing order. When the coefficient consists of a single a_j , the stride is defined as n , the size of the DFT.

In the polynomials above, the constant terms have base a_0 , the linear terms have base a_1 , and all four coefficients have stride 2. If the example above is continued to determine $f \bmod(x^8 - 1)$, each coefficient a_j will have base a_j and stride 8. For any k , the output value $X(k)$ has base a_0 and stride 1.

Definition 1.3. The *weight stride*, W_s , of a coefficient is the integer difference ($\bmod n$) of the powers put on ω_n to form the weights on any two successive a_j in the coefficient, when the terms of the coefficient are written with the indices in strictly increasing order.

The W_s of each coefficient in $f \bmod(x^2 - \omega_8^0)$ above is 0. The W_s of each coefficient in $f \bmod(x^2 - \omega_8^4)$ above is 4. For $f \bmod(x^8 - 1)$ in the example above, there is no *combination* of the a_j comprising any coefficient, so the W_s of each of the eight original coefficients is defined to be zero. The W_s for $X(k) = a_0\omega_n^0 + a_1\omega_n^k + a_2\omega_n^{2k} + \dots + a_{n-1}\omega_n^{(n-1)k}$ is k .

Definition 1.4. The *weight on base*, W_b , of a coefficient in an intermediate FFT result is the integer power ($\bmod n$) to which ω_n has been raised to form the accumulated weight on the base of the coefficient.

The W_b of each of the four coefficients in the example, indeed of any coefficient from the radix-2 FFT, is zero. To find an example of coefficients with nonzero W_b among the common FFTs, we'll consider the size-8 twisted FFT. Given $f(x)$ as in the example, the

remainder $f \bmod (x^4 + 1)$ determines the remainder $f(\omega_8 x) \bmod (x^4 - 1)$ as described in [1] and [42]. It follows that one of the intermediate results of the size-8 twisted FFT is

$$\begin{aligned} f(\omega_8 x) \bmod (x^2 - 1) &= (a_0 - a_4 + \omega_8^2[a_2 - a_6]) + (\omega_8^1[a_1 - a_5] + \omega_8^3[a_3 - a_7])x \\ &= (a_0 + a_2\omega_8^2 + a_4\omega_8^4 + a_6\omega_8^6) + (a_1\omega_8^1 + a_3\omega_8^3 + a_5\omega_8^5 + a_7\omega_8^7)x. \end{aligned}$$

So we see that the coefficient of the linear term has base a_1 and $W_b = 1$. This W_b as well as the other definitions from this section are visually summarized in Figure 1.

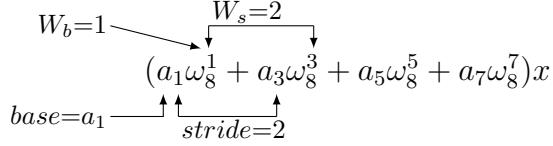


Figure 1. Definitions

2. A Flowgraph Representation for Generating a Family of FFT Algorithms

Signal flowgraphs are a widely used formalism to represent and analyze FFTs[10][5]. In this section we show how the concepts defined in Section 1.1 occur in flowgraphs of common power-of-two FFTs. In particular, we will show that each node in a given flowgraph can be labeled with a 3-tuple, $(\text{stride}, \text{base}, W_s)$, which is an invariant for a family of FFT algorithms that can be realized by that given flowgraph. This invariant can then be used to generate FFT instances realizable by that flowgraph.

To facilitate the discussion, we show two example flowgraphs. The first, shown in Figure 3, is Gauss' original FFT[25][1]. The second, shown in Figure 5, represents a size-16 conjugate split-radix as discussed in [32][33].

2.1 Edges and Nodes

Each directed edge represents the transfer of a complex number, either into or out of a node. In an algorithmic implementation of the flowgraph, each edge is indeed a single concrete complex number, but for the purposes of our flowgraph analysis, this complex number should be thought of symbolically as a weighted sum of the a_j , where the weight on any a_j is some ω_n^* .

The input operands of the FFT, labeled $a_0 \dots a_{n-1}$, are shown at the top and the output values, labeled $X(0) \dots X(n-1)$, are shown at the bottom. Unlike traditional FFT flowgraphs, we use a_j instead of $x(j)$ for input operands and show data flow top-to-bottom instead of left-to-right to facilitate discussion relating this flowgraph to the polynomial evaluation perspective of the FFT.

Each node represents complex addition and/or multiplication operations applied to the input operands to generate the output values. Figure 2 shows the internal behavior of a node. For nodes with two input edges, the two input operands are added to produce the single complex result id , when viewed concretely. We prefer to view id symbolically, as a weighted sum of the a_j , where the weight on any a_j is some ω_n^* . Next, id is separately

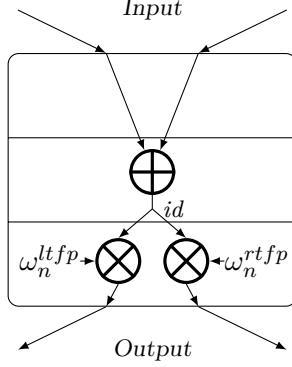


Figure 2. Node Internal Behavior

multiplied by two **twiddle factors** to produce the left and right output values. In Figure 2 the left and right twiddle factors are shown as ω_n^{ltfp} and ω_n^{rtfp} respectively but in the concise node representation as seen in Figure 3 only the integers $ltfp$ (left twiddle factor power) and $rtfp$ (right twiddle factor power) are shown in the bottom row of each node.

We count the cost of multiplication by some twiddle factor ω_n^{tfp} in the traditional way, where the cost of multiplication by $1, -1, i$ or $-i$ is free, multiplication by $\sqrt{i}, -\sqrt{i}, \sqrt{-i}$ or $-\sqrt{-i}$ is 4 floating point operations (FLOPs), and multiplication by any other n^{th} root of unity is 6 FLOPs. In addition to the potential multiplication cost, each node always requires 2 FLOPs for the cost of the addition.

There is one interesting multiplication cost exception when ω_n^{ltfp} and ω_n^{rtfp} are complex conjugates. In this case, $\Re(\omega_n^{ltfp}) = \Im(\omega_n^{rtfp})$ and $\Im(\omega_n^{ltfp}) = \Re(\omega_n^{rtfp})$ so that only 4 real multiplications and 4 real additions are needed to weight by both ω_n^{ltfp} and ω_n^{rtfp} . In this case, we tally 6 FLOPs for the weight by ω_n^{ltfp} and only an additional 2 FLOPs for the weight by ω_n^{rtfp} . For cases where $ltfp = rtfp$, we tally FLOPs for ω_n^{ltfp} only.

Two separate multiplications by ω_n^{ltfp} and ω_n^{rtfp} are never seen in traditional FFTs as it typically leads to higher cost when counting total floating point operations. Instead, one multiplication is done and the result may or may not be negated at no additional cost to generate the second output. In this paper, we adopt the more general description containing two separate multiplications and will later show how constraints can be applied to prune the search space to solutions requiring only a single multiplication without detriment to the final global FLOP count.

Dotted nodes in the top row of Figure 3 only have a single input operand and consequently there is no internal addition. In this case, the input operand is used directly as operand id within the node. Nodes in the bottom row of Figure 3 only have a single output value and consequently there is just a single internal multiplication. In this case, only a single twiddle factor is specified. Again, traditional FFTs often suppress this final multiplication as it is typically a cost-free multiplication by 1 or -1. For the generality of our first formulation, we always include this final multiplication, but will later prove via SMT that it is not required when searching for lowest FLOP count FFTs.

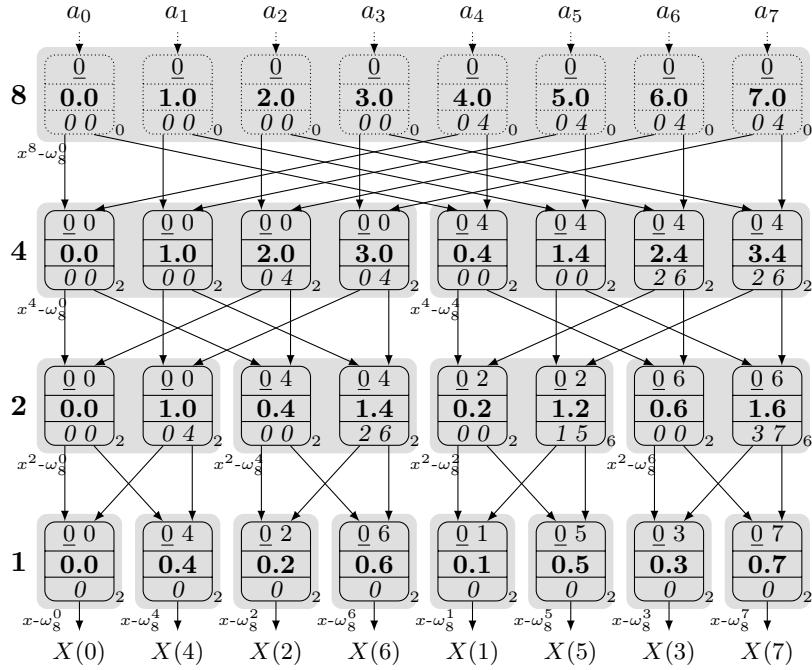


Figure 3. Size-8 Radix-2 DIF FFT Flowgraph

For the class of FFT flowgraphs we are considering, each node has at most two parents and two children. We adopt **dot notation** when it is necessary to refer to attributes of a node's parents or children. We refer to a node's left or right parent as $nd.lp$ or $nd.rp$, respectively, for a node nd . Likewise, $nd.lc$ or $nd.rc$ refer to a node's left or right child, respectively. With this notation, a node's left parent's left parent twiddle factor can be referred to concisely as $nd.lp.lp.\omega_n^{tfp}$. Note that tfp can be used here rather than lfp or rfp as it is clear from the graph context when tracing edges which twiddle factor applies.

2.2 Flowgraph Properties

Our flowgraph analysis requires that the following two properties be true, which are checked by computer traversal of the flowgraph.

Property 2.1. There is at most one path from any input operand a_j or internal node nd_p to any output value $X(k)$ or node nd_q .

Definition 2.1. The subset of input operands a_j that can reach a node contains that node's **original ancestors** and is denoted as $nd.A$ for a node nd . The subset of output values $X(k)$ that is reachable from a node contains that node's **terminal descendants** and is denoted as $nd.D$ for a node nd .

Property 2.2. For any node nd in the flowgraph, when the elements of $nd.A$ are ordered such that indices are strictly increasing, the difference (mod n) of indices on successive original ancestors in the list is constant. Furthermore, original ancestors of a node's left

parent interleave precisely with the original ancestors of the right parent, and the sets are always disjoint.

Example 2.1. For the node at the end of the third row in Figure 3 labeled with a bold 1.6,

$$nd.A = \{a_1, a_3, a_5, a_7\}$$

when ordered with strictly increasing indices. The integer difference (mod n) of successive original ancestor indices is always 2 for this example. For the node's left and right parents,

$$\begin{aligned} nd.A &= nd.lp.A \cup nd.rp.A \\ &= \{a_1, a_5\} \cup \{a_3, a_7\} \\ &= \{a_1, a_3, a_5, a_7\}, \end{aligned}$$

which interleave precisely when combined.

Flowgraphs adhering to these two properties are expected given the divide-and-conquer nature of common power-of-two FFTs. We have built flowgraphs of various size- n for radix-2, radix-4, decimation-in-time and decimation-in-frequency split-radix, conjugate split-radix [32] as well as twisted[1] FFTs, and have always found these properties to be true. For FFTs with some radix-4 content, this requires that when adding four numbers, the addition is factored into a binary addition tree that observes Property 2.2, which is what is commonly done. For twisted FFTs, different twisting functions ζ lead to different permutations of $X(k)$, but these are isomorphisms of the same flowgraph structure. It is not the point of this paper to prove which FFT algorithms generate which flowgraphs. Instead, we observe that many common power-of-two FFT algorithms generate flowgraphs that have these properties, and we require adherence to develop our flowgraph-based ideas.

2.3 A Node's *base* and *stride*

In the flowgraph shown in Figure 3, the left number in the middle row of each node is the *base* index for that node's *id* and the number at the left of an entire row of nodes is the *stride* for any node's *id*, when *id* is viewed symbolically as a weighted sum. Figure 4 is a key for all flowgraph labels and Figure 2 identifies the internal edge *id*.

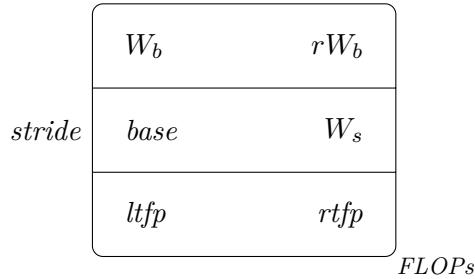


Figure 4. Flow Graph Node Key

Definition 2.2. A node's **base label**, $nd.\text{base}$, is always the index of the *base*, as defined in Definition 1.1, for the weighted sum $nd.\text{id}$ represented by the node. The number $nd.\text{base}$ is the minimum of $nd.\text{lp.base}$ or $nd.\text{rp.base}$. For the terminal case when nd has a single input operand, $nd.\text{base}$ is equal to j for the given input a_j .

To facilitate computation of $nd.\text{base}$, later computation of *weight on base*, as well as impose regularity on the flowgraph, the following property is enforced in flowgraph diagrams and computer data structures.

Property 2.3. For any node nd in a flowgraph, the relation ($nd.\text{lp.base} < nd.\text{rp.base}$) is always true.

Definition 2.3. A node's **stride label**, $nd.\text{stride}$ for a node nd , is always the *stride*, as defined in Definition 1.2, for the weighted sum $nd.\text{id}$ represented by the node. The number $nd.\text{stride}$ is the absolute difference (mod n) of $nd.\text{lp.base}$ and $nd.\text{rp.base}$. For the terminal case when nd has a single input operand, $nd.\text{stride}$ is defined to be n for a size- n FFT. Property 2.2 ensures that strides are constant and hence a single stride label per node is sufficient.

Example 2.2. From Example 2.1, we know that for the last node in the third row of Figure 3,

$$nd.A = \{a_1, a_3, a_5, a_7\}.$$

Ignoring values of applied weights in the flowgraph, the polynomial coefficient represented by this node must be of the form

$$nd.\text{id} = a_1\omega_8^* + a_3\omega_8^* + a_5\omega_8^* + a_7\omega_8^*.$$

From the discussion in Section 1.1 we can deduce that $nd.\text{base} = 1$ and $nd.\text{stride} = 2$. Also, we see that $nd.\text{lp.base} = 1$ and $nd.\text{rp.base} = 3$. By Definition 2.2,

$$\begin{aligned} nd.\text{base} &= \min\{nd.\text{lp.base}, nd.\text{rp.base}\} \\ &= \min\{1, 3\} \\ &= 1, \end{aligned}$$

and by Definition 2.3,

$$\begin{aligned} nd.\text{stride} &= nd.\text{rp.base} - nd.\text{lp.base} \pmod{n} \\ &= 3 - 1 \pmod{8} \\ &= 2. \end{aligned}$$

This node's row in the flowgraph is labeled with 2, the stride. The first label in the middle row of the node itself is 1, the base.

2.4 Weight on base

The *weight on base* for every node's input edge, as well as that node's weighted sum *id*, is recorded in the flowgraph. Even though W_b is defined for a true polynomial coefficient in

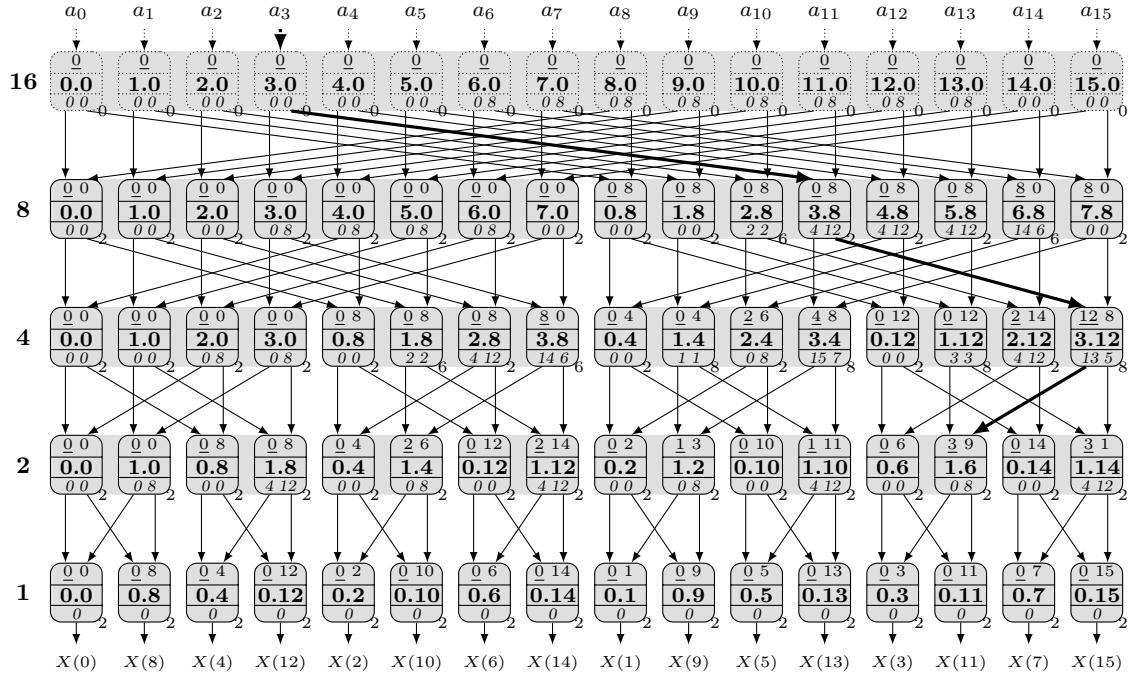


Figure 5. Size-16 Conjugate Split-Radix FFT

Definition 1.4, we record the *weight on base* for both the left and right input edge *before* the addition since both are required later to determine a node's weight stride. As shown in Figure 4, the top row of each node specifies W_b , the integer power (mod n) to which ω_n has been raised to form the accumulated weight on the *base* of the weighted sum of a_j represented by the left input edge. Likewise, rW_b represents the same for the right input edge. From Property 2.3, we know that after the addition the *base* of $nd.id$ is the same as the *base* of the left parent and that the addition does not alter weights. Thus, W_b for $nd.id$ is equal to the *weight on base* of the left input edge and there is no need for a separate lW_b .

Definition 2.4. The *weight on base* of a node's left input edge is,

$$nd.\omega_n^{W_b} = (nd.lparent.\omega_n^{tfp})(nd.lparent.\omega_n^{W_b}),$$

and likewise for a node nd 's right input edge is,

$$nd.\omega_n^{rW_b} = (nd.rparent.\omega_n^{tfp})(nd.rparent.\omega_n^{W_b}).$$

Following from Definition 2.2 and Property 2.3, the *weight on base* of $nd.id$ is equal to the *weight on base* of the left input edge and both are referenced as W_b . For the terminal case when nd has a single input, W_b is defined to be zero. Finally, note that W_b for all output values $X(k)$ is always zero as all $X(k)$ contain a constant term with a_0 that can only be weighted by ω_n^0 in any correct DFT.

Example 2.3. Since *weight on base* is the result of a series of multiplications by various roots of unity ω_n^* , it can also be viewed as addition (mod n) of the powers on the roots of unity. This is illustrated by the path shown with bold edges from input operand a_3 to a node nd with $nd.base = 1$ and $nd.stride = 2$ in Figure 5. Then

$$nd.\omega_{16}^{rW_b} = \omega_{16}^{13}(\omega_{16}^{12}(\omega_{16}^0(a_3))),$$

which is a_3 multiplied by all twiddle factors along the path, and can be rewritten as

$$\begin{aligned} rW_b &= 13 + 12 + 0 \pmod{16} \\ &= 9. \end{aligned}$$

This rW_b , 9, is shown in the upper right corner of the node. Once this path reaches nd , we no longer keep track of the weight on a_3 as it is no longer the *base* of the weighted sum *id*. However, it is still essential to keep track of this weight up to this point as it is used to compute W_s .

2.5 Weight Stride

A node's *weight stride* label, W_s , is shown at the right of each node's middle row, as seen in Figures 3, 4 and 5.

Definition 2.5. A node's *weight stride* label is

$$nd.W_s = nd.rW_b - nd.W_b \pmod{n}.$$

For the terminal case when nd has a single input, W_s is defined to be zero. The number $nd.W_s$ is always the W_s as defined in Definition 1.3 for the weighted sum $nd.id$ represented by the node.

Example 2.4. Again consider the node nd at the end of the bold path in Figure 5 where

$$\begin{aligned} nd.W_s &= nd.rW_b - nd.W_b \pmod{n} \\ &= 9 - 3 \pmod{16} \\ &= 6. \end{aligned}$$

This W_s , 6, appears as the last label in the middle row of this node. We can now reconstruct exactly the weighted sum of coefficient $nd.id$. For the node we are considering with *stride* = 2, *base* = 1, W_s = 6 and W_b = 3,

$$nd.id = a_1\omega_{16}^3 + a_3\omega_{16}^9 + a_5\omega_{16}^{15} + a_7\omega_{16}^5 + a_9\omega_{16}^{11} + a_{11}\omega_{16}^1 + a_{13}\omega_{16}^7 + a_{15}\omega_{16}^{13}$$

Now that a node's W_s is defined, we present a key observation that W_s is invariant across all FFTs that can be mapped to the given flowgraph. This invariance is central in defining a family of FFTs that can then be searched for desirable members.

Theorem 2.1. For a size- n FFT flowgraph constructed by *any* FFT algorithm such that Properties 2.1 and 2.2 hold, every node's W_s is an invariant.

Proof. Consider an arbitrary node nd in the flowgraph. Next, consider an arbitrary FFT output value from this node's terminal descendants, $X(k) \in nd.D$. For the two input values $a_{(nd.base)}$ and $a_{(nd.base)+(nd.stride)}$ in the weighted sum $X(k)$, it follows from the definition of the DFT that these input values must be weighted as

$$a_{(nd.base)}\omega_n^{k(nd.base) \pmod n} \text{ and } a_{(nd.base)+(nd.stride)}\omega_n^{k((nd.base)+(nd.stride)) \pmod n}.$$

Hence, the *weight stride* between these input values is

$$\begin{aligned} \text{weight stride} &= k((nd.base) + (nd.stride)) - k(nd.base) \pmod n \\ &= k(nd.stride) \pmod n. \end{aligned}$$

Since, by Property 2.1, nd is the only contributor of $a_{(nd.base)}$ and $a_{((nd.base)+(nd.stride))}$ to $X(k)$, and they are bound together by the addition in nd and never will be weighted again individually, we have

$$nd.W_s = k(nd.stride) \pmod n.$$

Any other value for $nd.W_s$ would produce an incorrect FFT result. \square

Example 2.5. Again consider the node at the end of the bold path in Figure 5 with $nd.W_s = 6$, $nd.stride = 2$ and $X(3), X(11) \in nd.D$. Then,

$$\begin{aligned} nd.W_s &= k(nd.stride) \pmod n \\ 6 &= 3 \times 2 \pmod{16} \\ &= 11 \times 2 \pmod{16}. \end{aligned}$$

2.6 Canonical Node Labels

Since the three node labels *base*, *stride* and W_s are either defined or proven to be unchanged by any applied weight ω_n , we can now assign a canonical label to each node. For a size- n FFT flowgraph, the set of canonical node labels defines a family of FFTs that can be realized by that flowgraph. Actual applied weights ω_n , interpreted as W_b , distinguish members in the family of FFTs.

Definition 2.6. A node's **canonical label** is $nd(nd.stride, nd.base, nd.W_s)$.

Example 2.6. Again consider the node at the end of the bold path in Figure 5. This node is labeled $nd(2, 1, 6)$ and is the only node with that label in the flowgraph. The $nd.stride = 2$ appears to the left of the row in which $nd(2, 1, 6)$ is found. The $nd.base = 1$ and $nd.W_s = 6$ appear in bold on the node itself.

2.7 Correspondence to the Polynomial View

Although our main representation is a flowgraph, we have relied on the polynomial view of the FFT to facilitate our discussion. In particular, each edge of the flowgraph represents a weighted sum of a_j and each $nd.id$ a coefficient of the original polynomial modulo one of its factors, also a weighted sum of a_j . We highlight with gray background in Figures 3 and

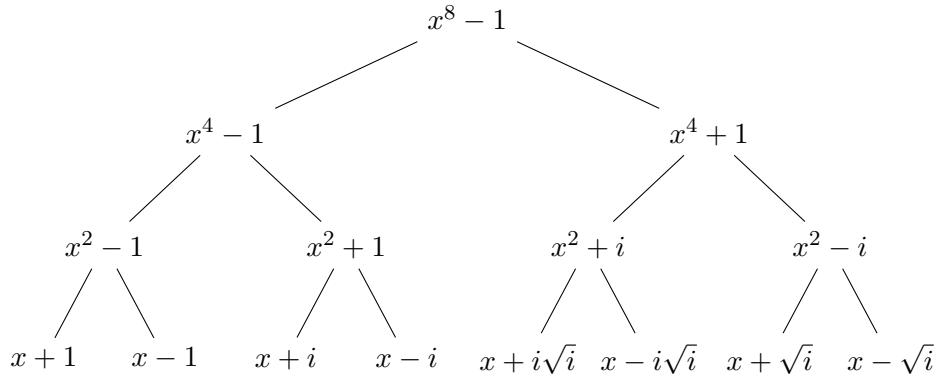


Figure 6. Factor lattice of $x^8 - 1$

the polynomial factor lattice as described by Bernstein in [1] and shown in Figure 6. The degree of each polynomial factor is the stride for all flowgraph nodes it contains. The power to which ω_n is raised to form the constant term in that polynomial is the W_{stride} for all flowgraph nodes it contains. And finally, each node's $nd.base$ is the index of lowest degree among the a_j used in the weighted sum represented by that node.

Recall our example for the case $n = 8$. We associate the sampled data to coefficients of a degree 7 polynomial:

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7.$$

This polynomial is an element of $\mathbb{C}[x]$, where we have a division algorithm that gives $f(x) \equiv r(x) \bmod p(x)$ when $f(x) = q(x)p(x) + r(x)$, for $r(x)$ of lesser degree than $p(x)$. In particular, we have $f(x) \equiv f(x) \bmod (x^8 - 1)$ because f has degree strictly less than 8. The residue class of f modulo $(x^8 - 1)$ determines the residue class of f modulo $(x^4 - 1)$ and modulo $(x^4 + 1)$, as the latter two polynomials are factors of $x^8 - 1$. Given a complete factorization of $x^8 - 1$ into distinct, irreducible, linear factors as shown in Figure 6,

we have the following isomorphism of rings:

$$\mathbb{C}[x]/(x^8 - 1) \cong \mathbb{C}[x]/(x - 1) \times \mathbb{C}[x]/(x + 1) \times \mathbb{C}[x]/(x + i) \times \cdots \times \mathbb{C}[x]/(x + \sqrt{i}) \times \mathbb{C}[x]/(x - \sqrt{i}),$$

which follows from the Chinese Remainder Theorem, as seen in [19].

So an FFT algorithm is finding an element from the product ring that corresponds to the given $f(x) \in \mathbb{C}/(x^8 - 1)$. Our flowgraph highlights the path of the inputs through the lattice of factor rings and canonical homomorphisms in Figure 7. The intermediate polynomials are

$$\begin{aligned}
f(x) \bmod (x^4 - 1) &= (a_0 + a_4) + (a_1 + a_5)x + (a_2 + a_6)x^2 + (a_3 + a_7)x^3 \\
f(x) \bmod (x^4 + 1) &= (a_0 - a_4) + (a_1 - a_5)x + (a_2 - a_6)x^2 + (a_3 - a_7)x^3 \\
f(x) \bmod (x^2 - 1) &= (a_0 + a_4 + a_2 + a_6) + (a_1 + a_5 + a_3 + a_7)x \\
f(x) \bmod (x^2 + 1) &= (a_0 + a_4 - a_2 - a_6) + (a_1 + a_5 - a_3 - a_7)x \\
f(x) \bmod (x^2 - i) &= (a_0 - a_4) + (a_2 - a_6)i + [(a_1 - a_5) + (a_3 - a_7)i]x \\
f(x) \bmod (x^2 + i) &= (a_0 - a_4) - (a_2 - a_6)i + [(a_1 - a_5) - (a_3 - a_7)i]x.
\end{aligned}$$

Since finding the residue of $f(x) \bmod (x^4 + 1)$ is equivalent to setting x^4 equal to $-1 = \omega_8^4$ in $f(x)$, we see in the coefficients of $f(x) \bmod (x^4 + 1)$ pairs of the original inputs whose indices differ by 4. Viewing these coefficients as weighted sums of the a_j , where the a_j are written with the indices increasing, we note that successive weights change by a factor of $\omega_8^4 = -1$. Since finding the residue of $f(x) \bmod (x^2 + i)$ is equivalent to setting x^2 equal to $-i = \omega_8^2$ in $f(x)$, we see in the coefficients of $f(x) \bmod (x^2 + i)$ four of the original inputs whose indices differ by 2 when listed in increasing order. Viewing these coefficients as weighted sums of the a_j , where the a_j are written with the indices increasing, we note that successive weights change by a factor of $\omega_8^2 = -i$.

Example 2.7. In Figure 3, the $stride = 4$ row has two polynomials highlighted, the left labeled $x_8^4 - \omega_8^0$ and the right labeled $x_8^4 - \omega_8^4$. The right polynomial, $x_8^4 - \omega_8^4$, has four nodes corresponding to the four terms of this new polynomial. The constant term has $base = 0$, the linear term has $base = 1$, and so on, until the last node with $base = 3$ represents the coefficient of the x^3 term in the polynomial. Since these four nodes arise from $x_8^4 - \omega_8^4$, all nodes in this polynomial have $stride = 4$. Finally, since the constant term of the factor polynomial is $-\omega_8^4$, written as ω_n raised to a power instead of the usual $+1$, all nodes in this polynomial have a $W_{stride} = 4$.

This correspondence exists for the original FFT attributed to Gauss[25]. Twisting as described in [1] implies that we use a different factor lattice for $x^8 - 1$. But it is essential

$$\begin{array}{c}
f(x) \in \mathbb{C}/(x^8 - 1) \\
\downarrow \\
\mathbb{C}/(x^4 - 1) \times \mathbb{C}/(x^4 + 1) \\
\downarrow \\
\mathbb{C}/(x^2 - 1) \times \mathbb{C}/(x^2 + 1) \times \mathbb{C}/(x^2 + i) \times \mathbb{C}/(x^2 - i) \\
\downarrow \\
\mathbb{C}/(x - 1) \times \mathbb{C}/(x + 1) \times \cdots \times \mathbb{C}/(x + \sqrt{i}) \times \mathbb{C}/(x - \sqrt{i})
\end{array}$$

Figure 7. Factor rings with canonical homomorphisms

to remember that our flowgraph analysis to derive canonical labels is independent of any twists and will derive the same canonical labels for a size- n flowgraph regardless of what twists are applied in the particular FFT used to generate the flowgraph.

In the case of twisting $x^4 + 1$ to $x^4 - 1$ in a size-8 FFT, we see that

$$\begin{aligned} f(x) &\equiv r(x) \bmod(x^4 + 1) \\ \implies f(x) &= q(x)(x^4 + 1) + r(x) \\ \implies f(\omega_8 x) &= q(\omega_8 x)((\omega_8 x)^4 + 1) + r(\omega_8 x) \\ &= q(\omega_8 x)(\omega_8^4 x^4 + 1) + r(\omega_8 x) \\ &= (-1)q(\omega_8 x)(x^4 - 1) + r(\omega_8 x) \\ \implies f(\dot{x}) &\equiv r(\dot{x}) \bmod(\dot{x}^4 - 1), \end{aligned}$$

where $\dot{x} = \omega_8 x$. Taking the polynomial view, the element $\dot{x}^4 - 1 \in \mathbb{C}[\dot{x}]$ has a factor tree isomorphic to that of $x^4 - 1 \in \mathbb{C}[x]$. Whereas the factor ring $\mathbb{C}[x]/(x^4 + 1)$ may be considered a 4-dimensional vector space over \mathbb{C} with basis $\{1, x, x^2, x^3\}$, the new factor ring $\mathbb{C}[\dot{x}]/(\dot{x}^4 - 1)$ has basis $\{1, \omega_8 x, (\omega_8 x)^2, (\omega_8 x)^3\}$. So the *stride* and W_{stride} exhibited by each set of highlighted nodes in the flowgraph is preserved.

2.8 Generating a Family Member FFT Algorithm

Because W_s is independent of any particular FFT's twiddle factors, we can use it as the basis for generating all members of a family of FFT algorithms represented by a given flowgraph. A valid FFT can be created by *randomly* picking integer W_b values for all nodes in the flowgraph. Given these choices for W_b , W_s determines values for rW_b for all nodes in the flowgraph. Next, W_b and rW_b determine values for all twiddle factors, and a unique assignment of twiddle factors distinguishes a member in the family of FFT algorithms. Before we present a more formal algorithm for this process, we must first define how twiddle factors can be determined from W_b .

Definition 2.7. Following from Definition 2.4, a node's **twiddle factors**, ω_n^{lfp} and ω_n^{rtfp} , can be determined from W_b :

$$\begin{aligned} (nd.lp.\omega_n^{lfp})(nd.lp.\omega_n^{W_b}) &= nd.\omega_n^{W_b} \\ nd.lp.\omega_n^{lfp} &= (nd.\omega_n^{W_b}) / (nd.lp.\omega_n^{W_b}). \end{aligned}$$

This can be expressed as (mod n) subtraction of powers:

$$nd.lp.tfp = nd.W_b - nd.lp.W_b \pmod{n}.$$

The twiddle factor for a right parent is similarly defined as:

$$nd.rp.tfp = nd.rW_b - nd.rp.W_b \pmod{n}.$$

Example 2.8. Consider the node $nd(2, 1, 6)$ in Figure 5. For this node, $nd.W_b = 3$ and $nd.rW_b = 9$ are specified. Also, $nd.lp.W_b = 0$ and $nd.rp.W_b = 12$ are specified. Hence,

$$\begin{aligned} nd.lp.tfp &= nd.W_b - nd.lp.W_b \pmod{n} \\ &= 3 - 0 \pmod{16} \\ &= 3, \end{aligned}$$

which is the twiddle factor applied to that edge by $nd.lp$ as shown in the Figure. Likewise,

$$\begin{aligned} nd.rp.tfp &= nd.rW_b - nd.rp.W_b \pmod{n} \\ &= 9 - 12 \pmod{16} \\ &= 13, \end{aligned}$$

which is the twiddle factor applied to that edge by $nd.rp$.

Algorithm 1: How to Generate a Random Member FFT Algorithm

Input: Size- n flowgraph with labeled invariants
Output: Size- n flowgraph with twiddle factors assigned

```

1 foreach  $nd \in \text{flowgraph}$  do
2   if  $nd.stride \neq n$  then
3      $nd.W_b \leftarrow \text{rand}() \pmod{n}$ 
4      $nd.rW_b \leftarrow nd.W_b + nd.W_s \pmod{n}$ 
5   else
6      $nd.W_b \leftarrow 0$ 
7 foreach  $nd \in \text{flowgraph}$  do
8   if  $nd.stride \neq n$  then
9      $nd.lp.tfp \leftarrow nd.W_b - nd.lp.W_b \pmod{n}$ 
10     $nd.rp.tfp \leftarrow nd.rW_b - nd.rp.W_b \pmod{n}$ 
11   if  $nd.stride = 1$  then
12      $nd.tfp \leftarrow 0 - nd.W_b \pmod{n}$ 
```

Example 2.9. Figure 8 shows a random member from the family of FFTs realizable by a size-8 flowgraph. Consider node $nd(1, 0, 3)$. Since $nd.stride \neq n$, we assign a random integer ($\pmod{8}$) of 3 to $nd.W_b$. Following Algorithm 1,

$$\begin{aligned} nd.rW_b &= nd.W_b + nd.W_s \pmod{n} \\ &= 3 + 3 \pmod{8} \\ &= 6. \end{aligned}$$

This same process is repeated until W_b and rW_b have been assigned for all nodes with $stride \neq n$ in the flowgraph. Nodes with a single input, shown as dotted, always have $W_b = 0$ following Definition 2.4. Next, actual twiddle factors are computed from the weight assignments. Again consider node $nd(1, 0, 3)$ and the computation of twiddle factors for

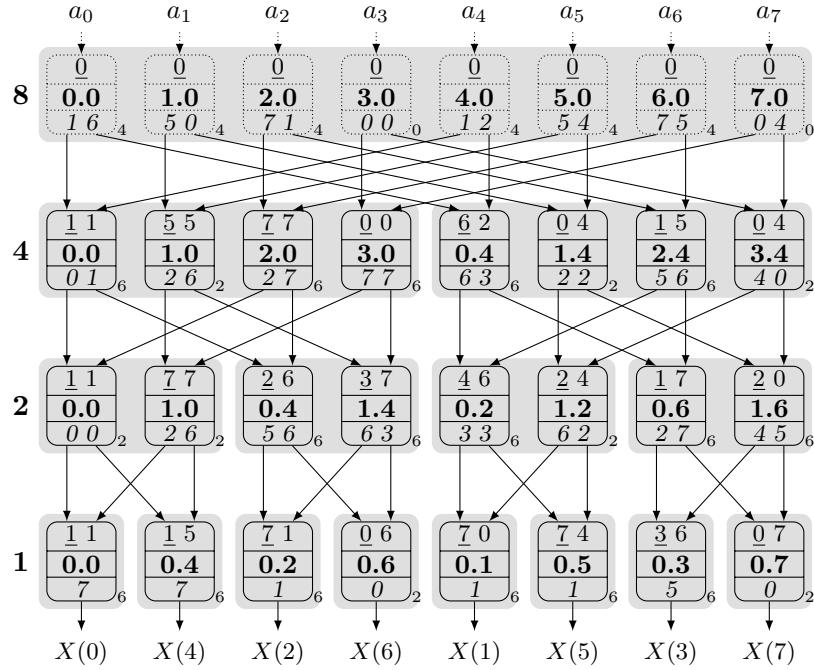


Figure 8. Size-8 Random FFT Flowgraph

that node's parents:

$$\begin{aligned}
 nd.lp.ltfp &= nd.W_b - nd.lp.W_b \pmod{n} \\
 &= 3 - 1 \pmod{8} \\
 &= 2 \\
 nd.rp.ltfp &= nd.rW_b - nd.rp.W_b \pmod{8} \\
 &= 6 - 2 \pmod{8} \\
 &= 4.
 \end{aligned}$$

Since nd has no children nodes ($nd.stride = 1$) we must compute its twiddle factor as

$$\begin{aligned}
 nd.tfp &= 0 - nd.lp.W_b \pmod{n} \\
 &= 0 - 3 \pmod{8} \\
 &= 5.
 \end{aligned}$$

This process is repeated until all twiddle factors are assigned.

3. Searching a Family of FFT Algorithms

In Section 2, a flowgraph representation of common power-of-two FFTs is developed that defines an invariant *weight stride*, W_s , for each node in the flowgraph. Algorithm 1 uses

the *weight stride* invariants to generate a new assignment of twiddle factors and hence a unique FFT. Since Algorithm 1 is arbitrary, a rich solution space of valid FFTs, called a family, results. In this section, we characterize the size of this family, specify a family as a Satisfiability Modulo Theory (SMT) problem, and demonstrate SMT solver-based search of this solution space. Although this search can be directed in various ways, we use it to prove the lowest total arithmetic complexity (fewest required FLOPs) when all twiddle factors are n^{th} roots of unity.

3.1 The Size of a Family

The solution space of valid FFTs for a given flowgraph is extremely large!

Definition 3.1. A size- n flowgraph's **solution cardinality** is $2^{n \log_2 n \log_2 n}$, and is the number of valid FFTs realizable by the given flowgraph. By direct examination of Algorithm 1, each node nd in a size- n flowgraph, where $nd.stride \neq n$, is arbitrarily assigned some integer (mod n) to $nd.W_b$. Thus, there are $n = 2^{\log_2 n}$ possible choices for a single node's W_b . And, since there are $n \log_2 n$ nodes in the flowgraph where $nd.stride \neq n$, there are $(2^{\log_2 n})^{n \log_2 n} = 2^{n \log_2 n \log_2 n}$ possible assignments of all W_b in a size- n flowgraph.

Example 3.1. For a size-256 flowgraph, there are

$$\text{solution cardinality} = 2^{256 \log_2 256 \log_2 256} = 2^{16384}$$

valid FFTs possible. One can better appreciate the magnitude of this number when reminded that the estimated number of atoms in the universe is 2^{264} and the current fastest supercomputer performs 2^{144} FLOPs per second. Yet this number is very small when compared to all valid and invalid n^{th} root of unity assignments possible for twiddle factors, 2^{34816} . Thus, for this size-256 flowgraph, there is just a 1 in 2^{18432} chance of guessing correct twiddle factors.

Although the solution space is immense, in practice we are only interested in family members with desirable qualities, such as fewest required FLOPs, better precision¹, improved performance or ease of implementation on a specific microarchitecture. Consequently, we need a way to search this space and find these more desirable family members.

3.2 A First SMT Formulation

Because of the way concepts were developed in Section 2, it is straight-forward to model Algorithm 1 as an SMT problem. This is best illustrated by considering Listing 1, which shows portions of the SMT model in SMT-LIB 1.2 format[49] that is created to find a lowest arithmetic complexity instance of a size-16 FFT. After a standard preamble, lines 4 and 5 declare the external inputs $nd(2, 1, 6).W_b$ and $nd(2, 1, 14).W_b$, which are both 4-bit vectors. Although not shown in the listing, inputs for all undetermined W_b are included. It is for these variables that the SMT solver attempts to find a satisfying assignment. For nodes where $nd.stride = n$, the value $nd.W_b$ is predetermined to be 0 and is declared as a constant.

1. All family members are exact and do not sacrifice numerical accuracy. Imprecision arises from choice of twiddle factors with values very close to zero and consequent floating-point representation limitations.

An example of this is shown in line 9 and corresponds to Algorithm 1 line 6. Next, rW_b for all nodes is computed via addition of W_b and $nd.stride$. An instance of this is seen in line 11 and corresponds to Algorithm 1 line 4. Note that all addition and subtraction is naturally $(\text{mod } n)$ given the fixed-size bitvectors in the SMT formulation. Twiddle factors for all nodes are computed as illustrated in lines 13 and 14. This corresponds to lines 9-12 of Algorithm 1.

Unlike Algorithm 1, the objective of the SMT model is to find the lowest arithmetic cost. For this, we must compute the cost implied by every twiddle factor. Lines 16-18 show the computation of cost predicates $c0, c4, c6$, (0, 4 or 6 FLOPs for multiplication, respectively), for the left twiddle factor of $nd(4, 1, 12)$. Not shown in this listing are any necessary predicates $c0, c2, c4, c6$ for multiplication cost incurred by the right twiddle factor. Line 19 shows cost predicates used in an if-then-else (ITE) tree to compute the multiplication FLOPs required by a node. We compute predicates first and then a numeric cost as the predicates are useful in defining pruning constraints later. In line 21, a total cost is computed by simply adding up all node multiplication costs. Finally, line 22 constrains the total cost to be less than or equal to some constant, and line 23 specifies that this multiplication FLOP constraint is satisfied. Note that the FLOP count due to a node's addition is constant for the flowgraphs under consideration and is not explicitly included in the SMT models.

```

1 (benchmark example1
2 :logic QF_BV
3 ...
4 :extrafuns ((Wb_2_1_6 BitVec[4]))
5 :extrafuns ((Wb_2_1_14 BitVec[4]))
6 ...
7 :formula
8 ...
9 (let (?Wb_16_14_0 bv0[4])
10 ...
11 (let (?rWb_2_1_6 (bvadd Wb_2_1_6 bv6[4]))
12 ...
13 (let (?ltfp_4_1_12 (bvsb Wb_2_1_6 ?Wb_4_1_12))
14 (let (?ltfp_4_3_12 (bvsb ?rWb_2_1_6 ?Wb_4_3_12))
15 ...
16 (flet ($c0_4_1_12 (= (extract [1:0] ?ltfp_4_1_12) bv0[2]))
17 (flet ($c4_4_1_12 (and (= (extract [0:0] ?ltfp_4_1_12) bv0[1]) (not $c0_4_1_12)))
18 (flet ($c6_4_1_12 (not (= (extract [0:0] ?ltfp_4_1_12) bv0[1])))
19 (let (?cost_4_1_12 (ite $c6_4_1_12 bv6[4] (ite $c4_4_1_12 bv4[4] bv0[4])))
20 ...
21 (let (?totalcost (bvadd ?cost_2_2_1 (bvadd ?cost_4_1_12 ?cost_4_3_12)) ...
22 (flet ($maxcost (bvule ?totalcost bv22[4]))
23 $maxcost
24 )...)
```

Listing 1. Sample SMT Code

In practice, more care is given to the total cost addition seen in line 21. A balanced adder tree is constructed, where each add uses only as many bits as required for the worst case. Furthermore, following the recursive structure in the FFT apparent from the polynomial view, cost for smaller FFTs are computed first and then combined to compute the cost for larger parent FFTs. This total cost computation is effectively a pseudo-Boolean constraint, and we have tried implementing it as an if-then-else (ITE) tree similar to the ROBDD-techniques described in [21]. Our experience is that the adder tree is 2-3 times better

in terms of SMT computation time for this particular problem with the Boolector SMT solver[7]. We did not implement the sorter-based technique described in [21].

To find the lowest arithmetic complexity, the SMT model is repeatedly solved, each time with a lower value for the constant seen in line 22. At some point the model becomes unsatisfiable and the lowest possible arithmetic complexity is known. Unfortunately, this straight-forward implementation does not scale up. For flowgraphs of size-32, the time for computing the unsatisfiability of a 455 FLOP solution requires 30 seconds using the Boolector solver [7] on a 64-bit Intel Core i7 Linux machine. At size-64 and for unsatisfiable cost of 1159 FLOPs, we reach our timeout of 24 hours without determining unsatisfiability.

3.3 Cost Symmetries

As formulated so far, the SMT model supports the full range of possible values for each twiddle factor since each twiddle factor is modeled as a size- m bitvector. This much information is not necessary for finding the lowest possible arithmetic complexity, and only adds to the complexity of the model. Instead, it is possible to express every twiddle factor as

$$\omega_n^{tfp} = \omega_n^{tfp - (tfp \pmod{n/4})} \omega_n^{tfp \pmod{n/4}}.$$

In this expression, $\omega_n^{tfp - (tfp \pmod{n/4})}$ specifies the quadrant in which ω_n^{tfp} lies and is always a free multiplication by 1, $-1, i$ or $-i$. Consequently, the portion of ω_n^{tfp} that solely contributes to multiplication cost can be represented by just a quarter of the n^{th} roots of unity and is defined as $\psi_n^{tfp} = \omega_n^{tfp \pmod{n/4}}$. To simplify the SMT model and upcoming partitioning, we suppress the quadrant rotations, $\omega_n^{tfp - (tfp \pmod{n/4})}$, and only reason with ψ_n .

Multiplication of two ψ_n is well defined and can be expressed as modular arithmetic. Consider the multiplication

$$\omega_n^{a+b \pmod{n}} = \omega_n^a \omega_n^b,$$

which can be re-expressed as

$$\omega_n^{a+b-(a+b \pmod{n/4}) \pmod{n}} \psi_n^{a+b \pmod{n/4}} = \omega_n^{a-(a \pmod{n/4})} \psi_n^a \omega_n^{b-(b \pmod{n/4})} \psi_n^b.$$

If all ω_n specifying quadrant rotations are ignored, multiplication of two ψ_n is just

$$\psi_n^{a+b \pmod{n/4}} = \psi_n^a \psi_n^b,$$

which can be expressed easily using modular arithmetic in the SMT model.

From the bitvector perspective, suppressing ω_n quadrant rotations means that the two most significant bits of every weight on base, W_b or rW_b , need not be included in the SMT model. The SMT solver finds a satisfying assignment for all but the two most significant bits of every weight on base. The two most significant bits are then picked at random as done in Algorithm 1 without altering cost. In the end, all bits must be assigned to realize a correct FFT.

Eliminating these cost symmetries in the SMT model reduces a size- n flowgraph's solution cardinality to $2^{n \log_2 n ((\log_2 n) - 2)}$. For a size-256 flowgraph, this is a substantial reduction in the size of the solution space from 2^{16384} to 2^{12288} . Computation time for proving that a size-32 flowgraph has no solution with total cost equal to or less than 455 FLOPs is now 27 seconds. The timeout of 24 hours is still reached for a size-64 flowgraph constrained to 1159 FLOPs. It is possible that the SMT computation time improves only modestly since the SMT solver is detecting these cost symmetries without explicit help.

3.4 Butterflies

The next three techniques to simplify and partition the SMT model require reasoning with FFT butterflies. Although FFT butterflies are a well established idea, we define and review concepts relevant to our SMT model.

Definition 3.2. A size- q **butterfly** is any subgraph of a size- n FFT flowgraph that is graph isomorphic to a size- q FFT flowgraph where $q \leq n$. A butterfly's **canonical label** is $bf(nd.stride, nd.base, nd.W_s, q)$ for the single node $nd \in bf$ such that $nd.stride, nd.base$ and $nd.W_s$ are less than or equal to the $stride, base$ and W_s of any other node in the butterfly. As with all FFT flowgraphs considered in this paper, q must be some power of two.

Example 3.2. In Figure 5, the butterfly $bf(1, 0, 0, 2)$ contains the four nodes $nd(1, 0, 0)$, $nd(1, 0, 8)$, $nd(2, 0, 0)$ and $nd(2, 1, 0)$. The expected traditional butterfly structure is clearly seen with the node used for identification, $nd(1, 0, 0)$, at the bottom left. This same node, $nd(1, 0, 0)$, is also used to identify the size-4 butterfly $bf(1, 0, 0, 4)$ which contains 12 nodes and is also clearly visible. Less obvious are small butterflies that appear toward the top of the flowgraph such as $bf(4, 2, 0, 2)$ which contains the four nodes $nd(4, 2, 0)$, $nd(4, 2, 8)$, $nd(8, 2, 0)$ and $nd(8, 6, 0)$. The larger butterfly $bf(4, 2, 0, 4)$ which contains 12 nodes can also be traced with $nd(4, 2, 0)$ anchoring the bottom left corner. Finally, the entire flowgraph in Figure 5 can be denoted as $bf(1, 0, 0, 16)$.

It is also useful to refer to nodes in an arbitrary butterfly not by canonical node label but by relative position. To facilitate this, one can view the nodes of a butterfly as forming a matrix and use matrix row,column indexing to refer to a specific node, $nd_{r,c}$. For example, for any size-2 butterfly, the top-left corner node is $nd_{0,0}$, the top-right corner node is $nd_{0,1}$, the bottom-left corner node is $nd_{1,0}$, and the bottom-right corner node is $nd_{1,1}$.

Property 3.1. For size-2 and size-4 butterflies in the flowgraph, all W_s for nodes in the same row are congruent modulo $n/4$. The value to which they are all congruent modulo $n/4$ is referred to as $nd_{r,*}.W_s$. This property arises from the correspondence of W_s in a flowgraph to the polynomial view as described in Section 2.7.

Example 3.3. Consider the size-4 butterfly $bf(1, 0, 3, 4)$ from Figure 5. By inspection,

$$\begin{aligned} nd_{0,*}.W_s &= \{12, 12, 12, 12\} && \equiv 0 \pmod{n/4} \\ nd_{1,*}.W_s &= \{6, 6, 14, 14\} && \equiv 2 \pmod{n/4} \\ nd_{2,*}.W_s &= \{3, 11, 7, 15\} && \equiv 3 \pmod{n/4}. \end{aligned}$$

3.5 Shared Twiddle Factors

Our formulation permits two multiplications, by ω_n^{lfp} and ω_n^{rtfp} , per node in the FFT flowgraph. Although this generality may be useful for some algorithms, we show here that it is not needed when minimizing the total FLOP count is the objective. In fact, it only increases the complexity of the SMT model.

Theorem 3.1. For any size-2 butterfly, bf_1 , such that left and right twiddle factors are unshared per node in row 0 ($nd_{0,c}.lfp \neq nd_{0,c}.rtfp$) but shared per node in row 1

$(nd_{1,c}.ltfp = nd_{1,c}.rtfp)$ there exists another size-2 butterfly, bf_2 , such that left and right twiddle factors are shared per node for all nodes, that realizes all final weighted sums $X(0)$ and $X(1)$ possible by bf_1 . Furthermore, no bf_1 exists with lower FLOP count than some bf_2 .

Proof. The proof is in two parts. First, we prove the existence of three different bf_2 . Consider the computation performed by bf_1 ,

$$X(0) \equiv \psi_n^{nd_{1,0}.tfp} (a_0 \psi_n^{nd_{0,0}.ltfp} + a_1 \psi_n^{nd_{0,1}.ltfp}) \pmod{n/4}, \quad (1a)$$

$$X(1) \equiv \psi_n^{nd_{1,1}.tfp} (a_0 \psi_n^{nd_{0,0}.rtfp} + a_1 \psi_n^{nd_{0,1}.rtfp}) \pmod{n/4}. \quad (1b)$$

Because of Property 3.1 and Theorem 2.1, $nd_{0,*}$ left and right twiddle factors are related by a common weight stride,

$$\psi_n^{nd_{1,*}.W_s} \equiv \frac{\psi_n^{nd_{0,1}.W_b} \psi_n^{nd_{0,1}.ltfp}}{\psi_n^{nd_{0,0}.W_b} \psi_n^{nd_{0,0}.ltfp}} \equiv \frac{\psi_n^{nd_{0,1}.W_b} \psi_n^{nd_{0,1}.rtfp}}{\psi_n^{nd_{0,0}.W_b} \psi_n^{nd_{0,0}.rtfp}} \pmod{n/4}.$$

This simplifies to

$$\frac{\psi_n^{nd_{0,1}.ltfp}}{\psi_n^{nd_{0,0}.ltfp}} \equiv \frac{\psi_n^{nd_{0,1}.rtfp}}{\psi_n^{nd_{0,0}.rtfp}} \pmod{n/4}, \quad (2)$$

which directly relates left and right twiddle factors for nodes in row 0 of bf_1 .

Equations 1 and 2 can be used to derive three different bf_2 , labeled bf_{2A} , bf_{2B} and bf_{2C} . Butterfly bf_{2A} , with $nd_{0,0}.ltfp = nd_{0,0}.rtfp = 0$, is created by multiplying Equation 1a by $1 = \frac{\psi_n^{nd_{0,0}.ltfp}}{\psi_n^{nd_{0,0}.rtfp}}$ and Equation 1b by $1 = \frac{\psi_n^{nd_{0,0}.rtfp}}{\psi_n^{nd_{0,0}.ltfp}}$,

$$X(0) \equiv (\psi_n^{nd_{1,0}.tfp} \psi_n^{nd_{0,0}.ltfp}) (a_0 \frac{\psi_n^{nd_{0,0}.ltfp}}{\psi_n^{nd_{0,0}.ltfp}} + a_1 \frac{\psi_n^{nd_{0,1}.ltfp}}{\psi_n^{nd_{0,0}.ltfp}}) \pmod{n/4}$$

$$X(1) \equiv (\psi_n^{nd_{1,1}.tfp} \psi_n^{nd_{0,0}.rtfp}) (a_0 \frac{\psi_n^{nd_{0,0}.rtfp}}{\psi_n^{nd_{0,0}.rtfp}} + a_1 \frac{\psi_n^{nd_{0,1}.rtfp}}{\psi_n^{nd_{0,0}.rtfp}}) \pmod{n/4}.$$

After simplification of twiddle factors, the two twiddle factors applied to a_0 are now shared (ψ_n^0) and the two twiddle factors applied to a_1 are also shared due to Equation 2.

Butterfly bf_{2B} , with $nd_{0,0}.rtfp$ made equal to $nd_{0,0}.ltfp$, is created by multiplying Equation 1b by $1 = \frac{\psi_n^{nd_{0,0}.ltfp} \psi_n^{nd_{0,0}.rtfp}}{\psi_n^{nd_{0,0}.ltfp} \psi_n^{nd_{0,0}.rtfp}}$,

$$X(0) \equiv \psi_n^{nd_{1,0}.tfp} (a_0 \psi_n^{nd_{0,0}.ltfp} + a_1 \psi_n^{nd_{0,1}.ltfp}) \pmod{n/4}$$

$$X(1) \equiv (\psi_n^{nd_{1,1}.tfp} \frac{\psi_n^{nd_{0,0}.rtfp}}{\psi_n^{nd_{0,0}.ltfp}}) (a_0 \frac{\psi_n^{nd_{0,0}.ltfp} \psi_n^{nd_{0,0}.rtfp}}{\psi_n^{nd_{0,0}.rtfp}} + a_1 \frac{\psi_n^{nd_{0,0}.ltfp} \psi_n^{nd_{0,1}.rtfp}}{\psi_n^{nd_{0,0}.rtfp}}) \pmod{n/4}.$$

Again from direct inspection and application of Equation 2, every node shares left and right twiddle factors.

Butterfly bf_{3C} , with $nd_{0,0}.ltfp$ made equal to $nd_{0,0}.rtfp$, is created by multiplying Equation 1a by $1 = \frac{\psi_n^{nd_{0,0}.ltfp} \psi_n^{nd_{0,0}.rtfp}}{\psi_n^{nd_{0,0}.rtfp} \psi_n^{nd_{0,0}.ltfp}}$,

$$X(0) \equiv (\psi_n^{nd_{1,0}.ltfp} \frac{\psi_n^{nd_{0,0}.ltfp}}{\psi_n^{nd_{0,0}.rtfp}})(a_0 \frac{\psi_n^{nd_{0,0}.ltfp} \psi_n^{nd_{0,0}.rtfp}}{\psi_n^{nd_{0,0}.ltfp}} + a_1 \frac{\psi_n^{nd_{0,1}.ltfp} \psi_n^{nd_{0,0}.rtfp}}{\psi_n^{nd_{0,0}.ltfp}}) \pmod{n/4}$$

$$X(1) \equiv \psi_n^{nd_{1,1}.ltfp} (a_0 \psi_n^{nd_{0,0}.rtfp} + a_1 \psi_n^{nd_{0,1}.rtfp}) \pmod{n/4}.$$

Here, too, every node shares left and right twiddle factors.

Second, exhaustive search with SMT is used to prove that no bf_1 exists with lower FLOP count than some bf_2 . The SMT-based proof is a miter between bf_1 and bf_{2A} , bf_{2B} and bf_{2C} . The bf_1 side of the miter is a size-2 FFT modeled in SMT as described in Section 3.2. Additional constraints that $nd_{1,0}.ltfp = nd_{1,0}.rtfp$ and $nd_{1,1}.ltfp = nd_{1,1}.rtfp$ are added for bf_1 . The bf_2 side of the miter includes models for all three cases A, B and C. These are also modeled in SMT as described in Section 3.2 but with the additional constraint that each node has just one twiddle factor, tfp . Furthermore, the constraints $bf_{2A}.nd_{0,0} = 0$, $bf_{2B}.nd_{0,0}.tfp = bf_1.nd_{0,0}.ltfp$, and $bf_{2C}.nd_{0,0}.tfp = nd_{0,0}.rtfp$ are included with the respective bf_2 models. Input values a_j with arbitrary initial weights on base $nd_{0,c}.W_b$ and row weight strides $nd_{1,*}.W_s$ are common to all bf_1 and bf_2 . The free variables decided by the SMT solver include these common initial weights on base and row weight strides as well as weights on base per node for all row 1 nodes in bf_1 and bf_2 . FLOP counts for bf_1 , bf_{2A} , bf_{2B} and bf_{2C} , are individually and explicitly tallied within the SMT model. The question posed to the SMT solver is to find a bf_1 with lower FLOP count than bf_{2A} , bf_{2B} or bf_{2C} . The theorem is proved for some n if the SMT solver returns unsatisfiable. The proof can be run once for every size- n FFT flowgraph under consideration, or induction can be used to establish the result for $n + 1$ and higher. \square

Example 3.4. Consider the concrete computation performed by some bf_1 from a size-16 FFT flowgraph expressed as,

$$X(0) \equiv \psi_{16}^0 (a_0 \psi_{16}^1 + a_1 \psi_{16}^3) \pmod{n/4}$$

$$X(1) \equiv \psi_{16}^0 (a_0 \psi_{16}^3 + a_1 \psi_{16}^1) \pmod{n/4}.$$

By substituting into Equation 2, we see that this is a valid butterfly with weight stride adhering to Property 3.1,

$$\frac{\psi_{16}^3}{\psi_{16}^1} \equiv \frac{\psi_{16}^1}{\psi_{16}^3} \equiv \psi_{16}^2 \pmod{n/4}.$$

Some sharing can occur during the complex multiplication of a_0 and a_1 with these left and right twiddle factors since $\Re(\psi_{16}^1) = \Im(\psi_{16}^3)$ and $\Re(\psi_{16}^3) = \Im(\psi_{16}^1)$. Hence, the multiplication FLOP count for bf_1 is only $16 = 8 + 8 + 0 + 0$.

Butterfly bf_{2A} has only a ψ_{16}^0 twiddle factor applied to a_0 ,

$$X(0) \equiv \psi_{16}^1 (a_0 \psi_{16}^0 + a_1 \psi_{16}^2) \pmod{n/4}$$

$$X(1) \equiv \psi_{16}^3 (a_0 \psi_{16}^0 + a_1 \psi_{16}^2) \pmod{n/4}.$$

Note that the twiddle factors applied to a_0 and a_1 , ψ_{16}^0 , are shared for $X(0)$ and $X(1)$. The results for $X(0)$ and $X(1)$ are still equivalent to bf_1 as the final twiddle factors, $nd_{1,c}.tfp$, are now adjusted by factoring out ψ_{16}^1 and ψ_{16}^3 respectively. The total multiplication cost for bf_{2A} is $16 = 0 + 4 + 6 + 6$, which is the same as bf_1 .

Butterfly bf_{2B} has only a ψ_{16}^1 twiddle factor applied to a_0 ,

$$\begin{aligned} X(0) &\equiv \psi_{16}^0(a_0\psi_{16}^1 + a_1\psi_{16}^3) \pmod{n/4} \\ X(1) &\equiv \psi_{16}^2(a_2\psi_{16}^1 + a_1\psi_{16}^3) \pmod{n/4}. \end{aligned}$$

The ψ_{16}^2 is factored out of the sum in $X(1)$ to maintain equivalence with bf_1 . The total cost for bf_{2B} is also $16 = 6 + 6 + 0 + 4$.

Butterfly bf_{2C} has only a ψ_{16}^3 twiddle factor applied to a_0 ,

$$\begin{aligned} X(0) &\equiv \psi_{16}^2(a_0\psi_{16}^3 + a_1\psi_{16}^1) \pmod{n/4} \\ X(1) &\equiv \psi_{16}^0(a_2\psi_{16}^3 + a_1\psi_{16}^1) \pmod{n/4}. \end{aligned}$$

The ψ_{16}^2 is factored out of the sum in $X(0)$ to maintain equivalence with bf_1 . The total cost for bf_{2C} is also $16 = 6 + 6 + 4 + 0$. Although all butterflies in this example have the same multiplication cost, this is not always the case in general. The SMT portion of the proof of Theorem 3.1 shows that at least one case of bf_2 will have FLOP count less than or equal to bf_1 .

The definition of bf_1 in Theorem 3.1 requires that twiddle factors be shared per node in row 1, $nd_{1,c}.ltfp = nd_{1,c}.rtfp$. Butterflies meeting this constraint only occur at the bottom of the FFT flowgraph, where a single weight may be applied to some $X(k)$. But after applying Theorem 3.1 to all terminal size-2 butterflies in the bottom row, we now have shared twiddle factors in the next to the bottom row of the FFT flowgraph. Therefore, Theorem 3.1 can be applied iteratively to the entire flowgraph, starting from the bottom and proceeding to the top, so that all nodes have a single twiddle factor, tfp , without any FLOP count penalty.

Property 3.2. For any size-2 butterfly from a FFT flowgraph, if all nodes have a single shared twiddle factor tfp , then $nd_{1,0}.W_b \equiv nd_{1,1}.W_b \pmod{n/4}$. This is because $nd_{1,0}$ and $nd_{1,1}$ both have the same left parent with the same $tfp \pmod{n/4}$ applied.

Given Property 3.2, Algorithm 1 can now be updated so that the SMT formulation assigns a weight on base per size-2 butterfly and not per node. Instead of assigning a random W_b per node as seen in line 3 of the algorithm, a random W_b is assigned per bottom two nodes of every size-2 butterfly. This reduces the number of free W_b variables by half and substantially speeds up the SMT-based search. Shared twiddle factors in the SMT model reduce a size- n flowgraph's solution cardinality to $2^{\frac{n}{2}\log_2 n((\log_2 n)-2)}$. For a size-256 flowgraph, this further reduces the solution space to 2^{6144} . Computation time for proving that a size-32 flowgraph has no solution with total cost less than or equal to 455 FLOPs is now 3.5 seconds. The timeout of 24 hours is still reached for a size-64 flowgraph constrained to 1159 FLOPs.

3.6 Partitioning

Every SMT model formulated so far has been monolithic, and it has been computationally difficult to prove the lowest arithmetic complexity for any FFT larger than size-32. In this section, we show that analysis of butterflies at the top and bottom of the flowgraph can be used to partition larger FFTs into several smaller SMT models that can be solved. This analysis is facilitated by explicitly writing out the final weight on base computations, with all operations congruent ($\bmod n/4$), for an arbitrary size-4 butterfly:

$$\begin{aligned}
X(0).W_b &\equiv nd_{2,0}.tfp + nd_{1,0}.tfp + nd_{0,0}.tfp + nd_{0,0}.W_b \\
X(0).W_b &\equiv nd_{2,0}.tfp + nd_{1,0}.tfp + nd_{0,2}.tfp + nd_{0,2}.W_b - nd_{1,*}.W_s \\
X(0).W_b &\equiv nd_{2,0}.tfp + nd_{1,1}.tfp + nd_{0,1}.tfp + nd_{0,1}.W_b - nd_{2,*}.W_s \\
X(0).W_b &\equiv nd_{2,0}.tfp + nd_{1,1}.tfp + nd_{0,3}.tfp + nd_{0,3}.W_b - nd_{2,*}.W_s - nd_{1,*}.W_s \\
X(2).W_b &\equiv nd_{2,1}.tfp + nd_{1,0}.tfp + nd_{0,0}.tfp + nd_{0,0}.W_b \\
X(2).W_b &\equiv nd_{2,1}.tfp + nd_{1,0}.tfp + nd_{0,2}.tfp + nd_{0,2}.W_b - nd_{1,*}.W_s \\
X(2).W_b &\equiv nd_{2,1}.tfp + nd_{1,1}.tfp + nd_{0,1}.tfp + nd_{0,1}.W_b - nd_{2,*}.W_s \\
X(2).W_b &\equiv nd_{2,1}.tfp + nd_{1,1}.tfp + nd_{0,3}.tfp + nd_{0,3}.W_b - nd_{2,*}.W_s - nd_{1,*}.W_s \quad (3) \\
X(1).W_b &\equiv nd_{2,3}.tfp + nd_{1,3}.tfp + nd_{0,0}.tfp + nd_{0,0}.W_b \\
X(1).W_b &\equiv nd_{2,3}.tfp + nd_{1,3}.tfp + nd_{0,2}.tfp + nd_{0,2}.W_b - nd_{1,*}.W_s \\
X(1).W_b &\equiv nd_{2,3}.tfp + nd_{1,4}.tfp + nd_{0,1}.tfp + nd_{0,1}.W_b - nd_{2,*}.W_s \\
X(1).W_b &\equiv nd_{2,3}.tfp + nd_{1,4}.tfp + nd_{0,3}.tfp + nd_{0,3}.W_b - nd_{2,*}.W_s - nd_{1,*}.W_s \\
X(4).W_b &\equiv nd_{2,4}.tfp + nd_{1,3}.tfp + nd_{0,0}.tfp + nd_{0,0}.W_b \\
X(4).W_b &\equiv nd_{2,4}.tfp + nd_{1,3}.tfp + nd_{0,2}.tfp + nd_{0,2}.W_b - nd_{1,*}.W_s \\
X(4).W_b &\equiv nd_{2,4}.tfp + nd_{1,4}.tfp + nd_{0,1}.tfp + nd_{0,1}.W_b - nd_{2,*}.W_s \\
X(4).W_b &\equiv nd_{2,4}.tfp + nd_{1,4}.tfp + nd_{0,3}.tfp + nd_{0,3}.W_b - nd_{2,*}.W_s - nd_{1,*}.W_s.
\end{aligned}$$

All weights on base internal to the butterfly have been eliminated by repeated substitution. All weight strides for nodes in the same row are congruent due to Property 3.1. It is instructive to trace all 16 paths from an input operand to an output value for a size-4 butterfly and verify that the weight on base computation for that path is included in Equation 3.

3.6.1 PARTITIONING USING ORIGINAL BUTTERFLIES

At the top of a flowgraph, all a_j have a weight of 1, ω_n^0 . Butterflies that have input values which are some of these original a_j are called **original butterflies**. Analysis of original butterflies can exploit this known weight on a_j to partition the FFT flowgraph and hence the SMT model.

Property 3.3. The weight stride for all nodes in any butterfly that includes only nodes belonging to $f \bmod x^* - 1$, $x^* + 1$, $x^* - i$ and $x^* + i$ from the polynomial factor tree is congruent to 0 ($\bmod n/4$). This follows from the weight stride relationship to the polynomial view established in Section 2.7.

Example 3.5. Consider the size-4 original butterfly $bf(4, 2, 0, 4)$ from Figure 5. By inspection, W_s for all nodes in this butterfly $\{0, 4, 8, 12\}$ is congruent to 0 (mod 4). All size-4 original butterflies, and some larger, exhibit Property 3.3.

Theorem 3.2. For any arbitrary size-4 original butterfly, bf_1 , there exists another size-4 butterfly, bf_2 , which has zero-cost twiddle factors for nodes in rows 0 and 1, such that all realizable final weighted sums $X(k)$ of bf_1 can be realized by bf_2 . Furthermore, no bf_1 exists with lower FLOP count than this bf_2 .

Proof. The proof is in two parts. First, to prove all realizable final weighted sums of bf_1 can be achieved by bf_2 , we substitute 0 for all initial weights ($nd_{0,*}.W_b = 0$), for all twiddle factors in rows 0 and 1 ($nd_{0,*}.tfp = nd_{1,*}.tfp = 0$) and for all weight strides ($nd_{1,*}.W_s = nd_{2,*}.W_s = 0$), into the expressions from Equation 3:

$$\begin{aligned} X(0).W_b &\equiv nd_{2,0}.tfp + 0 + 0 + 0 \\ X(0).W_b &\equiv nd_{2,0}.tfp + 0 + 0 + 0 - 0 \\ X(0).W_b &\equiv nd_{2,0}.tfp + 0 + 0 + 0 - 0 \\ X(0).W_b &\equiv nd_{2,0}.tfp + 0 + 0 + 0 - 0 - 0 \\ X(2).W_b &\equiv nd_{2,1}.tfp + 0 + 0 + 0 \\ X(2).W_b &\equiv nd_{2,1}.tfp + 0 + 0 + 0 - 0 \\ X(2).W_b &\equiv nd_{2,1}.tfp + 0 + 0 + 0 - 0 \\ X(2).W_b &\equiv nd_{2,1}.tfp + 0 + 0 + 0 - 0 - 0 \\ X(1).W_b &\equiv nd_{2,3}.tfp + 0 + 0 + 0 \\ X(1).W_b &\equiv nd_{2,3}.tfp + 0 + 0 + 0 - 0 \\ X(1).W_b &\equiv nd_{2,3}.tfp + 0 + 0 + 0 - 0 - 0 \\ X(4).W_b &\equiv nd_{2,4}.tfp + 0 + 0 + 0 \\ X(4).W_b &\equiv nd_{2,4}.tfp + 0 + 0 + 0 - 0 \\ X(4).W_b &\equiv nd_{2,4}.tfp + 0 + 0 + 0 - 0 \\ X(4).W_b &\equiv nd_{2,4}.tfp + 0 + 0 + 0 - 0 - 0 \end{aligned}$$

By direct inspection we establish that any final weight on base can be realized by twiddle factors of nodes only in the last row.

Second, exhaustive search with SMT is used to prove that no bf_1 exists with lower FLOP count than its bf_2 . The SMT proof is a miter that includes bf_1 and bf_2 . The bf_2 side of the miter is a direct translation to SMT of the final weight on base computations just seen. The bf_1 side of the miter is created by substituting 0 for all initial weights ($nd_{0,*}.W_b = 0$) and for all weight strides ($nd_{1,*}.W_s = nd_{2,*}.W_s = 0$) in the expressions from Equation 3. Final weights $bf_1.X(k).W_b$ are required to be equivalent to corresponding final weights $bf_2.X(k).W_b$. FLOP counts for bf_1 and bf_2 are individually and explicitly tallied within the SMT model. The question posed to the SMT solver is to find a bf_1 with lower FLOP count than bf_2 . The theorem is proved for some n if the SMT solver returns unsatisfiable. The proof can be run once for every size- n FFT under consideration, or induction can be used to establish the result for $n + 1$ and higher. \square

This theorem appears to conflict with decimation-in-time FFTs, such as shown in Figure 5, where costly twiddle factors appear in the first two rows of the FFT. Consider the size-4 butterfly $bf(4, 2, 0, 4)$ from Figure 5. There is multiplication cost at internal nodes $nd(8, 2, 8)$ and $nd(8, 6, 8)$. But the twiddle factor, $nd(8, 2, 8).\omega_{16}^2$ can be factored out and pushed down to the children nodes $nd(4, 2, 4)$ and $nd(4, 2, 12)$. Likewise, an ω_{16}^2 must also be factored out of $nd(8, 6, 8)$ to maintain algebraic correctness. After factoring out the ω_{16}^2 , all multiplication cost occurs on the bottom row of $bf(4, 2, 0, 4)$ and the total cost remains 24 FLOPs. Globally, there is now no size-4 original butterfly with cost in the first two rows.

Because of Theorem 3.2 and the recursive structure of the FFT, we can now partition the FFT flowgraph when solving for minimum total arithmetic complexity. In general, we must solve for all FFTs corresponding to $f \bmod x^* - i$ and $x^* + i$ branches in the factor tree. For a size- n FFT, this requires solving SMT models for pairs of size- p butterflies, for all p from 1 up to $n/4$. In practice, for values of $p = 8$ the problem becomes trivial and is used as the terminal case of partitioning. The most difficult partition of a size- n FFT flowgraph, a size- $\frac{n}{4}$ butterfly, will have a solution space of $2^{\frac{n}{8} \log_2 \frac{n}{4} ((\log_2 n) - 2)}$. In more concrete terms, the largest SMT models required to solve a size-256 flowgraph are for two size-64 butterflies corresponding to the $f \bmod x^{64} - i$ and $x^{64} + i$ branches of the factor tree. One of these size-64 butterflies has a solution space of 2^{1152} .

Computation time for proving that a partitioned size-64 FFT flowgraph has no solution with total cost equal to or less than 1159 FLOPs is now 2.8 seconds. Our timeout of 24 hours is reached when attempting to prove that a size-128 FFT flowgraph has no solution with total cost equal to or less than 2824 FLOPs.

3.6.2 PARTITIONING USING TERMINAL BUTTERFLIES

At the bottom of a flowgraph, the weight on base required for each final result $X(k)$ is known. This enables analysis of **terminal butterflies**, or butterflies producing some final values of $X(k)$, so that the model may be further partitioned.

Theorem 3.3. For any arbitrary size-4 terminal butterfly, bf_1 , there exists another size-4 butterfly, bf_2 , which has zero-cost twiddle factors for nodes in rows 1 and 2, such that all realizable final weighted sums $X(k)$ of bf_1 can be realized by bf_2 . Furthermore, no bf_1 exists with lower FLOP count than this bf_2 .

Proof. The proof is in two parts. First, to prove all realizable final weighted sums of bf_1 can be achieved by bf_2 , we substitute 0 for all final weights ($X(k).W_b = 0$) and for all twiddle factors in rows 1 and 2 ($nd_{1,*}.tfp = nd_{2,*}.tfp = 0$) into the expressions from Equation 3:

$$\begin{aligned} X(0).W_b &= 0 \equiv 0 + 0 + nd_{0,0}.tfp + nd_{0,0}.W_b \\ X(2).W_b &= 0 \equiv 0 + 0 + nd_{0,0}.tfp + nd_{0,0}.W_b \\ X(1).W_b &= 0 \equiv 0 + 0 + nd_{0,0}.tfp + nd_{0,0}.W_b \\ X(4).W_b &= 0 \equiv 0 + 0 + nd_{0,0}.tfp + nd_{0,0}.W_b \\ X(0).W_b &= 0 \equiv 0 + 0 + nd_{0,2}.tfp + nd_{0,2}.W_b - nd_{1,*}.W_s \\ X(2).W_b &= 0 \equiv 0 + 0 + nd_{0,2}.tfp + nd_{0,2}.W_b - nd_{1,*}.W_s \\ X(1).W_b &= 0 \equiv 0 + 0 + nd_{0,2}.tfp + nd_{0,2}.W_b - nd_{1,*}.W_s \\ X(4).W_b &= 0 \equiv 0 + 0 + nd_{0,2}.tfp + nd_{0,2}.W_b - nd_{1,*}.W_s \end{aligned}$$

$$\begin{aligned}
X(0).W_b &= 0 \equiv 0 + 0 + nd_{0,1}.tfp + nd_{0,1}.W_b - nd_{2,*}.W_s \\
X(2).W_b &= 0 \equiv 0 + 0 + nd_{0,1}.tfp + nd_{0,1}.W_b - nd_{2,*}.W_s \\
X(1).W_b &= 0 \equiv 0 + 0 + nd_{0,1}.tfp + nd_{0,1}.W_b - nd_{2,*}.W_s \\
X(4).W_b &= 0 \equiv 0 + 0 + nd_{0,1}.tfp + nd_{0,1}.W_b - nd_{2,*}.W_s \\
X(0).W_b &= 0 \equiv 0 + 0 + nd_{0,3}.tfp + nd_{0,3}.W_b - nd_{2,*}.W_s - nd_{1,*}.W_s \\
X(2).W_b &= 0 \equiv 0 + 0 + nd_{0,3}.tfp + nd_{0,3}.W_b - nd_{2,*}.W_s - nd_{1,*}.W_s \\
X(1).W_b &= 0 \equiv 0 + 0 + nd_{0,3}.tfp + nd_{0,3}.W_b - nd_{2,*}.W_s - nd_{1,*}.W_s \\
X(4).W_b &= 0 \equiv 0 + 0 + nd_{0,3}.tfp + nd_{0,3}.W_b - nd_{2,*}.W_s - nd_{1,*}.W_s
\end{aligned}$$

Rows have been reordered to group common twiddle factors. By direct inspection we establish that a final weight on base of 0 for all $X(k)$ can be realized by twiddle factors of nodes only in the first row.

Second, exhaustive search with SMT is used to prove that no bf_1 exists with lower FLOP count than its bf_2 . The SMT proof is a miter that includes bf_1 and bf_2 . The bf_2 side of the miter is a direct translation to SMT of the final weight on base computations just seen. The bf_1 side of the miter is created by substituting 0 for all final weights ($X(k).W_b = 0$) in the expressions from Equation 3. Input values $nd_{0,*}.W_b$ and row weight strides, $nd_{1,*}.W_s, nd_{2,*}.W_s$, are common to bf_1 and bf_2 . FLOP counts for bf_1 and bf_2 are individually and explicitly tallied within the SMT model. The question posed to the SMT solver is to find a bf_1 with lower FLOP count than bf_2 . The theorem is proved for some n if the SMT solver returns unsatisfiable. The proof can be run once for every size- n FFT under consideration, or induction can be used to establish the result for $n + 1$ and higher. \square

This theorem appears to conflict with decimation-in-frequency FFT algorithms, such as shown in Figure 3, where costly twiddle factors appear in the last two rows of the FFT flowgraph. Consider the size-4 butterfly $bf(1, 0, 1, 4)$ from Figure 3. There is multiplication cost at internal nodes $nd(2, 1, 2)$ and $nd(2, 1, 6)$. But the twiddle factor, $nd(2, 1, 2).\omega_8^1$ can be distributed and pushed up to the parent nodes $nd(4, 1, 4)$ and $nd(4, 3, 4)$. Likewise, an ω_8^1 must also be factored out of $nd(2, 1, 6)$ to maintain algebraic correctness. Now all multiplication cost occurs in the top row of $bf(1, 0, 1, 4)$ and the total cost remains the same. Globally, there is now no size-4 terminal butterfly with cost in the last two rows. Note that for this small size-8 FFT, this new configuration of twiddle factors now fails conditions for partitioning by original butterflies as costly twiddle factors now occur in the top two rows. For this reason, combined original and terminal partitioning is only applicable to size-16 and larger FFT flowgraphs.

By Theorem 3.3 and the recursive structure of the FFT, we can now further partition the FFT flowgraph when solving for minimum FLOP count. In general, we must solve for all FFTs corresponding to $f \bmod x^* - i$ and $x^* + i$ branches in the factor tree, but now each branch can be partitioned into four smaller equally sized FFTs. For a size- n FFT, this requires solving SMT models for groups of 8 size- p butterflies for all p from 1 up to $\frac{n}{16}$. In practice, for values of $p = 8$ the problem becomes trivial and that is used as the terminal case of partitioning. The most difficult partition of a size- n FFT flowgraph, a size- $\frac{n}{16}$ butterfly, will have a solution space of $2^{\frac{n}{32} \log_2 \frac{n}{16} ((\log_2 n) - 2)}$. In concrete terms, the largest SMT models required to solve a size-256 flowgraph are eight size-16 butterflies corresponding to the f

$\bmod x^{64} - i$ and $x^{64} + i$ branches of the factor tree. One of these size-16 butterflies has a solution space of 2^{192} .

We can now prove the surprising result that size-256 FFTs exists which require only 6616 FLOPs, rather than the 6664 FLOPs required by the traditional split-radix, even when twiddle factors are of modulus one. Finding a 6616 FLOP algorithm requires 22 seconds to compute when the lowest cost constraint is used for each partition. Just over 5 seconds is required for the toughest size-16 partition. Of course, searching for the lowest cost in a partition requires repeated SMT runs and consequently the total search time is higher. To prove that no solution exists with FLOP count lower than 6616 requires 160 seconds total, with the toughest partition requiring just over 50 seconds.

3.7 Symmetry Reductions

We find that there are many FFTs with equivalent final FLOP count yet with different twiddle factor values. Prior work in twisting[1][42] indicates that this should be expected. In this section, we highlight two types of symmetry reduction that reduce SMT run times. Many local symmetry reduction constraints are possible and we experimented with dozens but found only these two to be of any significance.

3.7.1 3-NODE SYMMETRIES

A size-2 butterfly is **3-node symmetric** if 3 of its 4 nodes require 6 FLOPs for multiplication. Symmetries are eliminated by forcing $nd_{1,0}.tfp$ to have no multiplication cost.

Example 3.6. Consider a concrete computation performed by a size-2 butterfly from the size-32 FFT flowgraph expressed as

$$\begin{aligned} X(0) &\equiv \psi_{32}^0(a_0\psi_{32}^1 + a_1\psi_{32}^3) \pmod{n/4} \\ X(1) &\equiv \psi_{32}^7(a_0\psi_{32}^1 + a_1\psi_{32}^3) \pmod{n/4}. \end{aligned}$$

This butterfly requires $18 = 6 + 6 + 6$ FLOPs for multiplication. If the ψ_{32}^3 is factored out to “zero” the weight on a_1 and shared twiddle factors are preserved, these equations can be expressed as

$$\begin{aligned} X(0) &\equiv \psi_{32}^3(a_0\psi_{32}^6 + a_1\psi_{32}^0) \pmod{n/4} \\ X(1) &\equiv \psi_{32}^2(a_0\psi_{32}^6 + a_1\psi_{32}^0) \pmod{n/4}, \end{aligned}$$

with total multiplication cost of $18 = 6 + 6 + 6$ FLOPs again. Alternatively, if the ψ_{32}^1 is factored out to “zero” the weight on a_0 , these equations can be expressed as

$$\begin{aligned} X(0) &\equiv \psi_{32}^1(a_0\psi_{32}^0 + a_1\psi_{32}^2) \pmod{n/4} \\ X(1) &\equiv \psi_{32}^0(a_0\psi_{32}^0 + a_1\psi_{32}^2) \pmod{n/4}, \end{aligned}$$

with total multiplication cost of $12 = 6 + 6$ FLOPs. For the values in this example we find a cost benefit from factoring out the ψ_{32}^1 . We must be pessimistic and assume the worse case, $18 = 6 + 6 + 6$ FLOPs, since only one weight is guaranteed zero-cost. It is also possible to “zero” the weight on the $X(1)$ sum by distributing the ψ_{32}^7 and achieve the same 12 FLOP configuration.

In the SMT model, 3-node symmetric size-2 butterflies are detected and only those with zero multiplication cost for $nd_{1,0}$ are allowed. This is built by defining the following illegal condition,

$$nd_{1,0}.c6 \wedge ((nd_{0,0}.c6 \wedge nd_{0,1}.c6) \vee (nd_{0,0}.c6 \wedge nd_{1,1}.c6) \vee (nd_{0,1}.c6 \wedge nd_{1,1}.c6)),$$

for each size-2 butterfly and then requiring the inverse be satisfied in the SMT model.

We have verified with SMT-based proofs like those seen previously that this constraint doesn't increase the butterfly's FLOP count. As in the example, it may lead to a lower FLOP count if some node other than $nd_{1,0}$ has an applied weight of zero. We have formulated more complex constraints to detect these better cases early but found negligible speed-up in SMT runs. Instead, we rely on the cost-constraint described in Section 3.2 to eventually eliminate bad choices. Finally, if the SMT solver happens to choose the better placement of zero applied weight to begin with, the node is not 3-node symmetric (multiplication cost is less than 16 FLOPs) and no 3-node symmetric constraint will apply.

3.7.2 BOTTOM EQUAL-PAIR SYMMETRIES

A size-2 butterfly has **equal-pair symmetries** if nodes $nd_{1,0}$ and $nd_{1,1}$ have multiplication cost and equal twiddle factors, $nd_{1,0}.tfp = nd_{1,1}.tfp$. This symmetry is eliminated by requiring that these identical twiddle factors in row 1 be distributed to row 0 nodes of the butterfly.

Example 3.7. Consider a concrete computation performed by a size-2 butterfly from the size-32 FFT flowgraph expressed as

$$\begin{aligned} X(0) &\equiv \psi_{32}^3(a_0\psi_{32}^0 + a_1\psi_{32}^0) \pmod{n/4} \\ X(1) &\equiv \psi_{32}^3(a_0\psi_{32}^0 + a_1\psi_{32}^0) \pmod{n/4}. \end{aligned}$$

This butterfly requires $12 = 6 + 6$ FLOPs for multiplication. If the ψ_{32}^3 is distributed, these equations can be expressed as

$$\begin{aligned} X(0) &\equiv \psi_{32}^0(a_0\psi_{32}^3 + a_1\psi_{32}^3) \pmod{n/4} \\ X(1) &\equiv \psi_{32}^0(a_0\psi_{32}^3 + a_1\psi_{32}^3) \pmod{n/4}, \end{aligned}$$

with total multiplication cost of $12 = 6 + 6$ FLOPs again. Another example with initial multiplication cost in row 0 is

$$\begin{aligned} X(0) &\equiv \psi_{32}^3(a_0\psi_{32}^2 + a_1\psi_{32}^0) \pmod{n/4} \\ X(1) &\equiv \psi_{32}^3(a_0\psi_{32}^2 + a_1\psi_{32}^0) \pmod{n/4}, \end{aligned}$$

with total multiplication cost of $18 = 6 + 6 + 6$ FLOPs. After distributing the ψ_{32}^3 , this becomes

$$\begin{aligned} X(0) &\equiv \psi_{32}^0(a_0\psi_{32}^5 + a_1\psi_{32}^3) \\ X(1) &\equiv \psi_{32}^0(a_0\psi_{32}^5 + a_1\psi_{32}^3), \end{aligned}$$

with lower multiplication cost of $12 = 6 + 6$ FLOPs. For the values in this example we find a benefit but note that the final FLOP count is never worse than the initial as proved with SMT.

In the SMT model, bottom equal-pair symmetric butterflies are not allowed. This is built by defining the following illegal condition,

$$(\neg nd_{1,0}.c0) \wedge (nd_{1,0}.tfp = nd_{1,1}.tfp),$$

for each size-2 butterfly and then requiring the inverse be satisfied in the SMT model.

We have verified with SMT-based proofs like those seen previously that this symmetry reduction doesn't increase the butterfly's FLOP count. A similar constraint for top equal-pair symmetric butterflies can be formulated, and even applied in combination with the bottom equal-pair symmetric constraint with care, but we found negligible speed-up in SMT runs when doing so.

These two symmetry reduction constraints now bring the total time for finding a 6616 FLOP count solution for a size-256 FFT down to 8 seconds. To prove that no solution exists with less than 6616 FLOPs now requires 50 seconds. It is now possible to find a 15128 FLOP count solution for a size-512 FFT in about 11 hours. We gave up on attempts to find solutions better than 15128 FLOPs after spending more than 14 days. There were four partitions for which we could not prove unsatisfiable when applying a FLOP count constraint of the "best found less one." From experience, we suspect that a 15127 FLOP solution is most likely unsatisfiable given the dramatic increase in SMT solver run times.

4. Results and Experiments

Table 1 summarizes our results for SMT-based search of various size FFT flowgraphs. For size-256 FFTs and larger, we see that algorithms with FLOP count lower than the traditional split-radix do exist even when all twiddle factors have modulus one. We also show FLOP counts for the traditional split-radix and for the tangent FFT[32][1], where twiddle factors are scaled and hence not modulus one. As expected, the required SMT time quickly becomes intractable as larger FFTs are considered. Yet it is still instructive to consider FFTs of relatively small size as such FFTs appear in larger FFTs. Finally, we do not know the number of FFT algorithms meeting these minimum FLOP count constraints but do know that there are many. We did search for multiple solutions of a size-256 FFT flowgraph partition and found hundreds before terminating. These solutions have both different values and placement patterns for costly twiddle factors.

FFT Size	Tangent $ \omega_n^* = *$	Split-Radix $ \omega_n^* = 1$	SMT Search			
			$ \omega_n^* = 1$			
	FLOPs	FLOPs	FLOPs	time(s)	FLOPs	time(s)
32	456	456	456	1.4×10^{-1}	455	1.5×10^{-1}
64	1152	1160	1160	3.1×10^{-1}	1159	3.3×10^{-1}
128	2792	2824	2824	9.3×10^{-1}	2823	1.1×10^0
256	6552	6664	6616	8.3×10^0	6615	5.0×10^1
512	15048	15368	15128	3.9×10^4	15127?	$>1 \times 10^6$

Table 1. Lowest FLOP Counts Found by SMT Search

The times reported in Table 1 are for the FLOP bounds at the boundary between satisfiable and unsatisfiable. We search for this boundary using binary search akin to Newton’s method. We start with the best known FLOP bound for that size and class of FFT found in the literature, and divide that by 2. If that is satisfiable, we consider that the best known FLOP count and repeat. But if it is unsatisfiable, we choose a new FLOP bound half way between the unsatisfiable FLOP bound and the last known satisfiable bound and repeat. A complete search does require more time than seen in Table 1, but we find that FLOP counts far away from the boundary are solved relatively fast, whether they are satisfiable or unsatisfiable. Only when the boundary is approached do times increase dramatically. Furthermore, by imposing a timeout, we can skew the search to approach the boundary from the satisfiable side, where FLOP counts are successively becoming lower. This improves overall search performance as proving satisfiable cases is generally less costly than proving unsatisfiable cases.

The reduction in FLOP count of FFTs found by SMT search appears to accelerate for larger n when compared to the tangent FFT. Our size-256 solution has an advantage of 48 FLOPs when compared to the traditional split-radix FFT, whereas the tangent FFT has an advantage of 112 FLOPs. At this size, our FFT provides $48/112 = 0.429$ of the advantage of the tangent FFT. At size-512, this advantage is $240/320 = 0.75$. It is unclear if this approaches the tangent FFT advantage asymptotically, eventually surpasses it, or degrades. We suspect that the opportunities for optimization may be increasingly richer as partition sizes and the number of costly twiddle factors that they contain grow.

4.1 SMT QF_BV Solver Experiments

The results reported so far have all been generated using the SMT solver Boolector[7]. In this section, we present results for various SMT solvers, and identify some SMT solver characteristics best suited for our problem.

We use four representative benchmarks for our experiments. The first, **Sz256_6616**, is the hardest partition from a size-256 flowgraph with a 6616 FLOP bound and is known to be satisfiable. The second, **Sz256_6615**, is also the hardest partition from a size-256 flowgraph but with a 6615 FLOP bound and is known to be unsatisfiable. Likewise, the third and fourth benchmarks, **Sz512_15128** and **Sz512_15127**, are the hardest partitions from a size-512 flowgraph with 6616 and 6615 FLOP bounds respectively. Only **Sz512_15128** is known to be satisfiable. Whether benchmark **Sz512_15127** is satisfiable is unknown, but we suspect it is unsatisfiable.

For state-of-the-art SMT solvers, we use the top four SMT solvers in the QF_BV category, closed quantifier-free formulas over the theory of fixed-size bitvectors, from the SMT-2011 competition[9]: Z3[16], STP2[24], Boolector[7] as well as MathSat5[8] main and application configurations. We include two additional QF_BV solvers that performed well in earlier competitions: Beaver[31] and Yices[20]. For the SMT-2011 competition solvers, we used the binary executables and unmodified run scripts from the SMT-2011 competition web site[9]. For Beaver and Yices, we downloaded the latest available version from the web: Beaver 1.2.0.780 and Yices 2.0, build date of July 29, 2010, for x86_64-unknown-linux-gnu. Both Beaver and Yices were executed without any additional command line options. We updated our pretty printer to support SMT-LIB 2.0[49] for the four SMT-2011 competition

solvers. Beaver and Yices were given SMT-LIB 1.2 input. All experiments were run on a 64-bit Intel Core i7 Linux machine.

Results for our SMT solver experiments are shown in Table 2. We have ordered the results from best to worst performance on benchmark `Sz256_6615`, which we consider the most representative as all lowest FLOP searches must end with a proven unsatisfiable case. For this unsatisfiable case, all solvers perform in the same order of magnitude, with the worst performer requiring $2.5 \times$ the amount of time as the best performer. For the satisfiable cases, we see a larger variation in performance due to the rich set of solutions that exist and the chances that a particular solver’s search strategy will find one first. All SMT solvers reached the timeout of 24 hours without solving `Sz512_15127`.

Solver	Sz256_6616 SAT time(s)	Sz256_6615 UNSAT time(s)	Sz512_15128 SAT time(s)	Sz512_15127 Unknown
Beaver	2.0×10^0	7.6×10^0	6.2×10^2	Timeout
STP2	0.5×10^0	1.2×10^1	8.4×10^3	Timeout
Boolector	3.4×10^0	1.2×10^1	2.9×10^4	Timeout
MathSAT5 app	4.0×10^0	1.5×10^1	5.7×10^4	Timeout
MathSAT5 main	5.2×10^0	1.8×10^1	1.1×10^4	Timeout
Z3	2.8×10^0	1.9×10^1	2.1×10^4	Timeout
Yices	4.3×10^0	1.9×10^1	6.8×10^4	Timeout

Table 2. SMT Solver Performance

For Beaver, the best performing SMT solver on `Sz256_6615`, we also varied the command line options to test their effect. The most noticeable differences, although minor, came from disabling optimizations. Table 3 summarizes our findings. Constant propagation appears to be the most effective for our application. It should prove beneficial to incorporate constant propagation at the high-level when we generate our initial SMT models.

Solver	Sz256_6615 UNSAT time(s)
Beaver –disable-const	9.9
Beaver (disable all)	9.5
Beaver –disable-commute	8.1
Beaver –disable-assoc	7.8
Beaver –disable-non-linear	7.7
Beaver (enable all)	7.6

Table 3. Beaver Optimization Options

4.2 Bit-Blasting and SAT Solver Experiments

A common trait of the three best SMT solvers for our problem, Beaver, STP2 and Boolector, is that they focus on bitvector problems. All three perform bit-blasting and then use a SAT solver back-end to solve a traditional SAT problem. Furthermore, they pay close attention

to the circuit structure when optimizing and bit-blasting. All three incorporate AIGs, And-Invert Graphs[36], and rewriting of AIGs. Beaver and STP2 use the ABC[43] library to facilitate this. Furthermore, Beaver employs the SAT solver nlsat[30], which operates on AIGs natively, as its default back-end. Since this circuit-centric approach works well for our problems, this section presents experimental data on bit-blasting flows and SAT solver performance when considered separately.

For each of the three best SMT solvers for our problem we implemented four experimental bit-blasting flows. At a high-level, these four flows are:

1. SMT solver circuit representation to CNF with ABC
2. SMT solver circuit representation to CNF with ABC after ABC optimization for SAT
3. SMT solver circuit representation to CNF with AIGER
4. SMT solver circuit representation to CNF with AIGER after ABC optimization for SAT

Since each SMT solver’s native circuit representation is slightly different, we first standardized all circuit representations to AIGs. Beaver incorporates ABC and can generate AIGs natively. We generated an AIG using the command line options `beaver -no-solve -aig -aig-file=<file.aig> <file.smt>`. STP2 also incorporates ABC but has no working command line option to generate an AIG file. Since STP2 is distributed as source, we were able to add an option to generate an AIG output file of it’s internal circuit representation. Boolector has an option to dump expressions in BTOR format. We generated BTOR with that option and converted it to AIG using `synthebtor -m <file.btor> <file.aig>`, which is a tool provided with the Boolector package. Thus, the bit-blasted representation from all three solvers are standardized as AIGs.

In flows 1 and 2, CNF is generated from AIG by ABC using `write_cnf`. In flows 3 and 4, CNF is generated from AIG by using the tool `aigtocnf`, which is part of the AIGER package[3]. In flows 2 and 4, the ABC optimization for SAT command `drwsat` is executed 3 times to generate a simplified AIG.

We selected 7 SAT solvers that are readily available and either have performed well in recent SAT competitions[40][12] or are used in the back-end for Beaver, STP2 or Boolector already.

- glueminisat 2.2.5[45]
- simplifying minisat 2.2.0[22]
- cryptominisat 2.9.0[53]
- precosat 570[2]
- lingeling 276[2]
- clasp 2.0.2[35]
- nlsat 05102009[30]

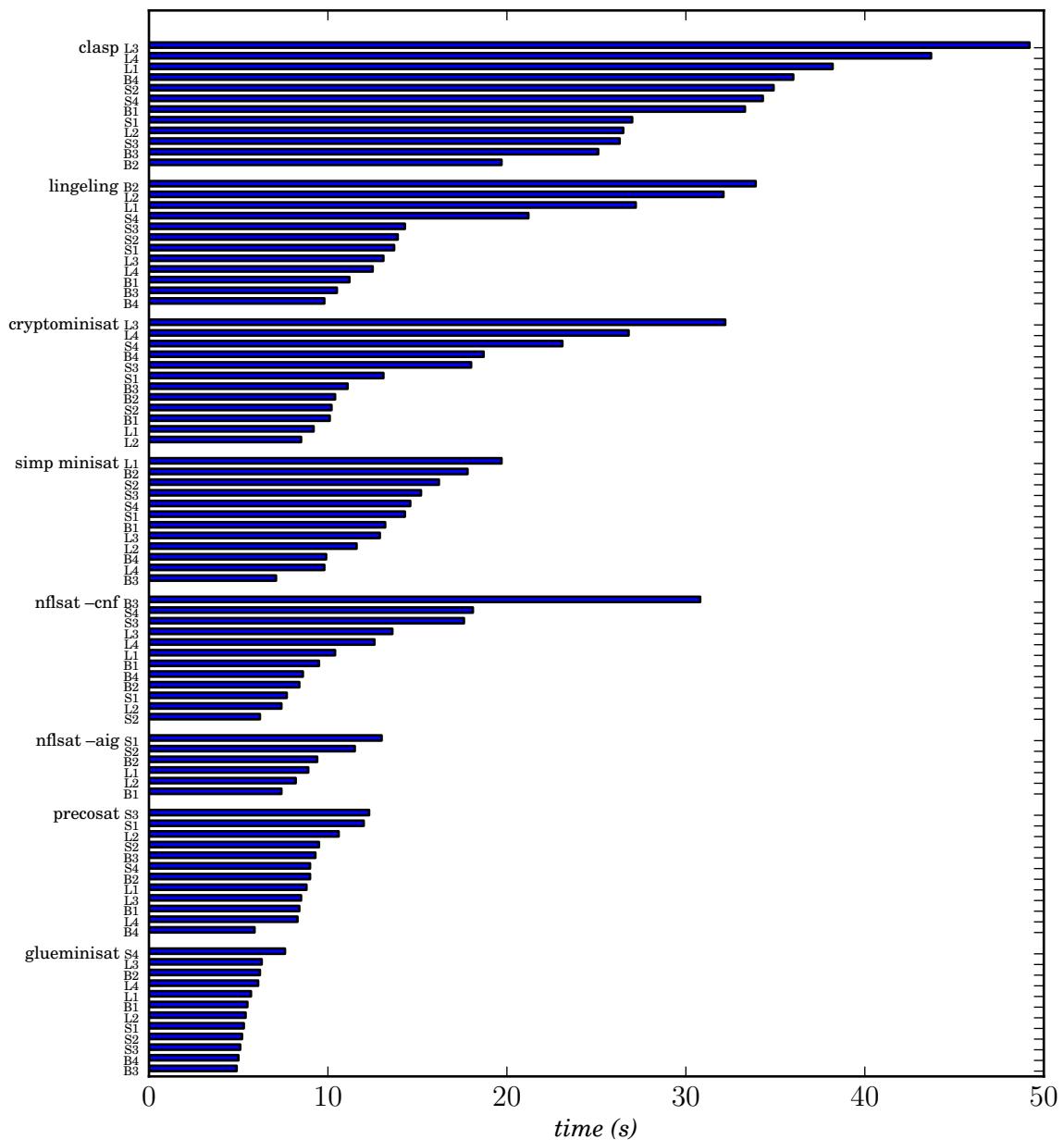


Figure 9. SAT Solver Performance for Various Bit-Blasting Flows

Figure 9 shows the results for each of the seven SAT solvers with all 12 bit-blasting flows. The bit-blasting flows are keyed to the SMT front-end, (B=Beaver, L=Boolector, S=STP2), along with the route to CNF, 1-4. Results are grouped by SAT solver, and ordered from worst (top) to best (bottom) average performance. Within a SAT solver grouping, results are ordered again from worst to best performance. The reported times include format conversion and ABC optimization, if applicable.

From these results, we see that the SAT solver glueminisat, the winner in the SAT 2011 competition for the UNSAT application track, consistently performs the best. Overall, the back-end SAT solver choice is more significant than the SMT bit-blasting tool and/or path to CNF. Still, Beaver bit-blasting appears to provide a slight second-order advantage. It is also interesting to note that the SAT solver clasp, the winner in the SAT 2011 competition for the UNSAT crafted track, performed the worst. Furthermore, Beaver’s default choice of a back-end SAT solver, nlsat with native AIG input, does not distinguish itself.

For our problem, the data suggests that bit-blasting by Beaver with conversion to CNF by AIGER (flow B3) for input to the SAT solver glueminisat is the best choice. We applied this flow to the unknown problem Sz512_15127 but were still unable to produce a result after days of compute time. From this we conclude that the greatest advances in solving our particular problem will come from high-level insight, such as additional problem partitioning and identification of new problem-specific constraints. Next, the choice of the underlying SAT technology will have some beneficial effect as SAT technology continues to improve. And finally, how we cast our problem as SAT, including initial specification, constant propagation, choice of bit-blasting and conversion to CNF, can be adjusted for further second-order improvements.

4.3 SMT QF_LIA Solver Experiments

If is unclear whether modeling our problem as QF_BV is the best choice. It is possible to model bitvector problems with other logics[26][57][6][4]. In this section, we present a first attempt to model our problem as QF_LIA, following the techniques of Kim and Somenzi[26].

In their recent paper[26], Kim and Somenzi showed that some QF_BV problems could be cast as QF_LIA for improved SMT solver performance. The main idea of their casting is to detect overflow and underflow of integer operations and use ITE operators to enforce the modular arithmetic of QF_BV within QF_LIA. We have implemented this casting by adding underflow/overflow detection and correction for all $(\text{mod } n)$ operations in Algorithm 1. Table 4 summarizes our results for all QF_LIA solvers we could find that accept SMT-LIB 2.0. The benchmarks Sz64_1160, Sz64_1159, Sz128_2824, Sz128_2823, are of similar character to the set used in QF_BV experiments shown in Table 2 except that they are from considerably smaller (size-64 and size-128) FFTs. When attempting the same benchmarks as used in the QF_BV experiments, the timeout of 24 hours was reached in all cases. Clearly, we must improve out initial QF_LIA specification and/or the underlying QF_LIA solver technology to make QF_LIA solvers competitive with QF_BV solvers on our problem.

4.4 Algorithm Design

The FFTs found by SMT-based search and posted on our web site[27] are witnesses that FFTs with lower total FLOP count than the split-radix exist even when all twiddle fac-

Solver	Sz64_1160 SAT time(s)	Sz64_1159 UNSAT time(s)	Sz128_2824 SAT time(s)	Sz128_2823 UNSAT time(s)
Z3	3.8×10^{-2}	5.2×10^{-2}	1.0×10^0	2.0×10^3
MathSAT5 app	7.6×10^{-2}	1.9×10^{-1}	1.5×10^3	Timeout
MathSAT5 main	5.5×10^{-2}	1.0×10^{-1}	1.0×10^2	1.4×10^3

Table 4. SMT QF_LIA Solver Performance

tors have modulus one, but are not practical algorithms in their current state. FFTs in widespread use usually can be defined succinctly in mathematical terms which leads to very regular patterns of twiddle factors in the FFT flowgraph. It is possible to formulate SMT constraints that require various forms of regularity in any satisfying solution. For example, additional constraints can be formulated and added to the model which allow costly twiddle factors only at specified nodes in the graph. A tighter constraint might force specific nodes to have prespecified twiddle factor values. A more relaxed constraint might just impose a relationship, such as a stride, between pairs of twiddle factors. In this way, the techniques described in this paper can be extended to do practical FFT algorithm design at the expense of proven optimality. Although this is a topic for further research, we highlight a few early experiments here.

The split-radix created by delayed twisting as described by Bernstein[1] and Mateer[42] is very succinct yet can be used to generate a rich family of highly regular split-radix algorithms simply by choosing different legal twisting coefficients, ζ . By examining the twiddle factor patterns generated by this algorithm, we determine that twiddle factors applied to ordered coefficients of a polynomial in the factor tree must have a constant stride (twisted), match constant values as seen in the classic decomposition (delayed twisting), or combine these two cases (twisting to something other than $x^* - 1$). With constraints formulated and applied to the SMT model that require this pattern of twiddle factors, we no longer find solutions with total FLOP count less than the split-radix for size-256 FFT flowgraphs. We do find solutions with FLOP count equal to the split-radix as expected. This confirms the theorem by Mateer[42] that combinations of twisting, though very rich, will never lead to an FFT with FLOP count lower than the split-radix. Although the regularity imposed by twisting doesn't support our solutions, other types of regularity might.

The tangent FFT[32][1] starts with a version of the conjugate split-radix FFT[33]. In this algorithm, twiddle factors occur as conjugate pairs, where the conjugate pair is either at the top or bottom of a size-2 butterfly. The complex twiddle factors for a conjugate pair can be factored as

$$\cos \alpha(1 + i \tan \alpha), \cos \gamma(1 + i \tan \gamma).$$

Since α and γ are conjugate angles, we know that $\cos \alpha = \cos \gamma$. Van Buskirk's trick[39], which is exploited in the tangent FFT, moves these real scaling factors so that their cost is absorbed by other multiplications. With constraints formulated and applied to the SMT model that require twiddle factors to occur globally as conjugate pairs, we no longer find solutions with total FLOP count less than the split-radix for size-256 FFT flowgraphs. We do still find instances with FLOP count equal to the split-radix. It still may be possible to

find solutions where conjugate pairs occur locally in specific places such that optimizations similar to Van Buskirk’s can be beneficially applied.

An objective to minimize FLOP count is primarily academic given the capabilities of modern computing hardware. Other more practical objectives include enhancing precision or easing implementation. For example, avoiding twiddle factors where the real or imaginary part is a number very close to zero may enhance the precision of the final result. Alternatively, restricting all twiddle factors to some limited set may ease implementation, and we can formulate a SMT model that does just that. There are size-32 FFTs that use just two non-trivial costly twiddle factors, plus the free multiplications by 1, -1 , i or $-i$. The minimum FLOP count for these algorithms is high at 616 compared to 456 for the split-radix but there may be benefits of having to multiply by just a few constants, especially in hardware implementations. If we increase the set of allowed non-trivial twiddle factors for a size-32 FFT to three, the minimum FLOP count is 536. For a size-64 FFT, we find a 2112 FLOP count solution that uses only non-trivial twiddle factor powers from the set $\{7, 8, 9\}$. Note that these twiddle factor powers include conjugates so that only three transcendental function computations or table look-ups are required. We have posted some examples of these FFTs on our web site[27].

5. Conclusions and Future Work

This paper presented a Boolean Satisfiability-based proof of the lowest FLOP count required by FFT algorithms up to size-512 with flowgraphs isomorphic to those generated by common power-of-two FFTs, and where all twiddle factors are n^{th} roots of unity. Even with these constraints, we find FFTs requiring fewer FLOPs than the split-radix starting at size-256. At the core of this proof is a novel way to enumerate all FFTs realizable by a given flowgraph. Partitioning and symmetry reduction techniques are developed to make it possible to prove FLOP count bounds for larger size-512 FFTs. Finally, because the SAT-based formulation and search techniques are general, the paper introduced additional search objectives that mimic twiddle factor patterns from twisting, require conjugate twiddle factor pairs, and minimize the allowed values of twiddle factors.

As seen from our experimental results, our biggest advances came from applying a high-level understanding of this problem to partition and detect symmetries in order to simplify the input for SMT and SAT solvers. We believe that more effort along these lines is a good direction for future work. In particular, work in symmetry detection and breaking to simplify SAT[51][34] is of interest. Just as this work uses computer automation to search for graph isomorphisms in the CNF structure, we can do the same at the more abstract FFT flowgraph level. Although symmetry breaking at the CNF level can benefit our problem, we believe that more progress can be made by exploiting higher-level symmetries in our specific problem. The challenge for us is to find useful isomorphisms with regard to twiddle factors, as the FFT flowgraph is very regular and rich in self-similarity. All our effort to partition and detect symmetry has been through human observation, and assistance from computer search may lead to better techniques.

We have cast finding FFT algorithms as a bitvector problem and have used SAT and SMT solvers in a stand-alone manner to find solutions. This raises two questions for future work. First, is QF_BV the best logic for this problem? Although we present preliminary

results when cast as QF_LIA in Section 4.3, there are still other casting techniques and logics to try[57][6][4][41]. Of particular interest to us is casting our problem to integer linear programming with the techniques presented by Brinkmann[6]. This may allow us to optimize larger problems, especially when optimality need not be proven. Second, can larger and more interesting instances of our problem be solved through tighter integration with SAT and/or SMT solvers? There are ideas for integrating optimization with SAT and SMT solvers[46][37][13]. Solvers such as STP2[24] are providing APIs for tighter integration of user’s applications. These directions remain unexplored by us but may yield significant improvements.

Besides the future work just described, we plan additional work in three more directions. First, Section 4.4 highlights FFT algorithm design possible with techniques described in this paper. We will study the applicability of our techniques to practical FFT algorithm design, with cost objectives ranging from improved precision to implementation on specific hardware[48]. Second, we seek to impose regularity on our lowest FLOP count solutions to determine if they can be described more traditionally as succinct algorithms. This should also help us better characterize the FLOP savings as the size of the FFT increases. Finally, we hope to ease the current constraint that all twiddle factors are n^{th} roots of unity, and thus incorporate optimizations similar to those in Van Buskirk’s[39] algorithm and the tangent FFT[32][1] directly into our search.

References

- [1] D. Bernstein. The tangent FFT. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 291–300, 2007.
- [2] A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Technical Report 10/1, Institute for Formal Models and Verification, Johannes Kepler University, 2010.
- [3] A. Biere. AIGER. <http://fmv.jku.at/aiger>, 2011.
- [4] M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzén, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani. Encoding rtl constructs for mathsat: a preliminary report. *Electronic Notes in Theoretical Computer Science*, **144**(2):3–14, 2006.
- [5] E.O. Brigham. *The fast Fourier transform and its applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
- [6] R. Brinkmann and R. Drechsler. Rtl-datapath verification using integer linear programming. In *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.*, pages 741–746. IEEE, 2002.
- [7] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177, 2009.

- [8] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The mathsat 4 smt solver. In *Computer Aided Verification*, pages 299–303. Springer, 2008.
- [9] R. Bruttomesso, M. Deters, and A. Griggio. SMTCOMP 2011. <http://www.satcomp.org/2011>, 2011.
- [10] C.S. Burrus. Appendix 1: FFT flowgraphs. <http://cnx.rice.edu/content/m16352/latest/>, September 2009.
- [11] C.S. Burrus. Polynomial description of signals. <http://cnx.rice.edu/content/m16327/latest/>, September 2009.
- [12] Sinz C. SAT-Race 2010. <http://baldur.iti.uka.de/sat-race-2010/index.html>, 2010.
- [13] A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability modulo the theory of costs: Foundations and applications. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–113, 2010.
- [14] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, **19**(90):297–301, 1965.
- [15] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, **5**(7):394–397, 1962.
- [16] L. De Moura, N. Bjørner, and C. Wintersteiger. Z3: Theorem Prover. <http://research.microsoft.com/en-us/um/redmond/projects/z3>, 2011.
- [17] P. Duhamel. Algorithms meeting the lower bounds on the multiplicative complexity of length- 2^n DFTs and their connection with practical algorithms. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, **38**(9):1504–1511, 2002.
- [18] P. Duhamel and M. Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing*, **19**(4):259–299, 1990.
- [19] D.S. Dummit and R.M. Foote. *Abstract Algebra*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [20] B. Dutertre and L. De Moura. The yices SMT solver. <http://yices.csl.sri.com>, 2010.
- [21] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**(3-4):1–25, 2006.
- [22] N. Eén and N. Sörensson. The MiniSat Page. <http://minisat.se>, 2011.
- [23] C.M. Fiduccia. Polynomial evaluation via the division algorithm the fast Fourier transform revisited. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 88–93. ACM, 1972.

- [24] V. Ganesh. STP Constraint Solver. <https://sites.google.com/site/stpfastprover/STP-Fast-Prover>, 2011.
- [25] C.F. Gauss. *Werke, Band 3*. Königlichen Gesellschaft der Wissenschaften, Göttingen, 1866. URL: http://134.76.163.65/agora_docs/41929TABLE_OF_CONTENTS.html.
- [26] Kim H., F. Somenzi, and H. Jin. Selective smt encoding for hardware model checking. In *Proceedings of the 9th International Workshop on Satisfiability Modulo Theories (SMT) 2011*, pages 49–58, 2011.
- [27] S. Haynal. Supporting Code Examples for Generating and Searching Families of FFT Algorithms. <http://www.softerhardware.com/fft>, 2011.
- [28] M. Heideman and C. Burrus. On the number of multiplications necessary to compute a length- 2^n DFT. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, **34**(1):91–95, 2003.
- [29] M.T. Heideman. *Multiplicative complexity, convolution, and the DFT*. Springer-Verlag New York, Inc. New York, NY, USA, 1988.
- [30] H. Jain and E.M. Clarke. Efficient sat solving for non-clausal formulas using dpll, graphs, and watched cuts. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 563–568. IEEE, 2009.
- [31] Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. 2009.
- [32] S.G. Johnson and M. Frigo. A modified split-radix FFT with fewer arithmetic operations. *Signal Processing, IEEE Transactions on*, **55**(1):111–119, 2007.
- [33] I. Kamar and Y. Elcherif. Conjugate pair fast Fourier transform. *Electronics Letters*, **25**(5):324–325, 1989.
- [34] H. Katebi, K. Sakallah, and I. Markov. Symmetry and satisfiability: An update. *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 113–127, 2010.
- [35] B. Kaufmann. clasp. <http://www.cs.uni-potsdam.de/clasp/>, 2011.
- [36] A. Kuehlmann, M.K. Ganai, and V. Paruthi. Circuit-based boolean reasoning. In *Design Automation Conference, 2001. Proceedings*, pages 232–237. IEEE, 2001.
- [37] J. Larrosa, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. A framework for certified boolean branch-and-bound optimization. *Journal of Automated Reasoning*, **46**(1):81–102, 2011.
- [38] D. Le Berre. SATLive! <http://www.satlive.org>, 2011.
- [39] T. Lundy and J. Van Buskirk. A new matrix approach to real FFTs and convolutions of length 2^k . *Computing*, **80**(1):23–45, 2007.

- [40] Järvisalo M., D. Le Berre, and O. Roussel. SAT Competition 2011. <http://www.satcompetition.org/2011>, 2011.
- [41] V. Manquinho, O. Roussel, and M. Deters. Pseudo-Boolean Competition 2011. <http://www.cril.univ-artois.fr/PB11>, 2011.
- [42] T. Mateer. *Fast Fourier transform algorithms with applications*. PhD thesis, Clemson University, 2008.
- [43] A. Mishchenko. ABC: A System for Sequential Synthesis and Verification. <http://www.eecs.berkeley.edu/~alanmi/abc>, 2011.
- [44] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference, 2001. Proceedings*, pages 530–535. IEEE, 2001.
- [45] H. Nabeshima, K. Iwanuma, and K. Inoue. GlueMiniSat. <https://sites.google.com/a/nabelab.org/glueminisat>, 2011.
- [46] R. Nieuwenhuis and A. Oliveras. On sat modulo theories and optimization problems. *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 156–169, 2006.
- [47] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM (JACM)*, **53**(6):937–977, 2006.
- [48] G. Nordin, P.A. Milder, J.C. Hoe, and M. Püschel. Automatic generation of customized discrete Fourier transform IPs. In *Proceedings of the 42nd annual Design Automation Conference*, pages 471–474. ACM, 2005.
- [49] S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. *Department of Computer Science, The University of Iowa, Tech. Rep*, 2006.
- [50] D.N. Rockmore. The FFT: An algorithm the whole family can use. *Computing in Science & Engineering*, **2**(1):60–64, 2002.
- [51] Karem A. Sakallah. Symmetry and satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, **185** of *Frontiers in Artificial Intelligence and Applications*, pages 289–338. IOS Press, 2009.
- [52] J.P.M. Silva and K.A. Sakallah. GRASP a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227. IEEE Computer Society, 1997.
- [53] M. Soos. CryptoMiniSat2. <http://www.msoos.org/cryptominisat2>, 2011.
- [54] H. Sorensen, M. Heideman, and C. Burrus. On computing the split-radix FFT. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, **34**(1):152–156, 2003.
- [55] S. Winograd. *Arithmetic complexity of computations*. Society for Industrial and Applied Mathematics, 1980.

- [56] R. Yavne. An economical method for calculating the discrete Fourier transform. In *Proc. AFIPS Fall Joint Computer Conf.*, **33**, pages 115–125. ACM, 1968.
- [57] Z. Zeng, P. Kalla, and M. Ciesielski. Lpsat: a unified approach to rtl satisfiability. In *Proceedings of the conference on Design, automation and test in Europe*, pages 398–402. IEEE Press, 2001.